(2)

DEPARTMENT OF INDUSTRIAL ENGINEERING

FAMU/FSU COLLEGE OF ENGINEERING

TALLAHASSEE, FLORIDA

Research Progress Report DASG 60/90/3

FINAL TECHNICAL REPORT ON

RESEARCH CONTRACT

#DASG 60-90-C-0142

Samuel Awoniyi, Ph.D.
Principal Investigator
(904)487-6354

Lester Frair, Ph.D.
Co-Principal Investigator
(904)487-6339

March 1991

This is the third of three technical reports for the USASDC research grant #DASG 60-90-C-0142. The first and second technical reports are included here as Appendix I and Appendix II respectively. The views, opinions, and/or findings contained in these reports are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless as designated by other official documentation.

DI-MISC-80048/M Scientific and Technial Reports Summary (Cont'd)
Block 10 PREPARATION INSTRUCTIONS (Con'd)

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Florida A&M University	6b. OFFICE SYMBOL (If applicable)
6c. ADDRESS (City, State, and ZIP Code): Tallahassee, Florida 32307	7a. NAME OF MONITORING ORGANIZATION: US Army SDC
	7b. ADDRESS (City, State, and ZIP Code): P.O. Box 1500 Huntsville, AL 35807-3801
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: US Army SDC	8b. OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): P.O. Box 1500 Huntsville, AL 35807-3801	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO
	DASG60-90-C-0142		

11. TITLE (Include Security Classification)

A Generalized Set-Covering Method for Surveillance Maintenance

12. PERSONAL AUTHOR(S)

Samuel Awoniyi, Ph.D. (PI) & Lester Frair, Ph.D. (Co-PI)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical Report	FROM 8/16/90 TO 3/15/91	March 14, 1991	85

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A heuristic procedure for the generalized set-covering problem is described in three technical reports. Computational experiments are presented, and a computer code (in Microsoft C) for the procedure is given. This heuristic procedure represents a new approach to discrete optimization - a nonlinear programming approach. This approach should be very useful in "solving" a host of NP-complete problems with nonlinear objective functions. To make the results of this research address real world, certain extensions are suggested. "Summary of Research Accomplishments" is included in the final report. A journal paper based on these research results is under preparation.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

SUMMARY OF RESEARCH ACCOMPLISHMENTS....3

SECTION 1: INTRODUCTION....4

SECTION 2: DESCRIPTION OF SOLUTION PROCEDURE....7

2.1 DETERMINING WHETHER SET IS COVERED....7

2.2 OUR PROCEDURE FOR PROBLEM (*)....9

SECTION 3: COMPUTATIONAL EXPERIMENTS....12

3.1 DESIGN CHOICES....12

3.2 HARDWARE AND PROGRAMMING....14

3.3 TEST PROBLEMS....15

SECTION 4: DIRECTIONS FOR FUTURE WORK....16

4.1 RESOURCE-CONSTRAINED EXTENSIONS....16

4.2 COMMITMENT-CONSTRAINED EXTENSIONS....17

4.3 MULTIPLE-LAYER SURVEILLANCE COVERAGE....17

REFERENCES....19

APPENDIX I - FIRST PRELIMINARY TECHNICAL REPORT....21

APPENDIX II - SECOND PRELIMINARY TECHNICAL REPORT....22

APPENDIX III- SOURCE CODE FOR COMPUTER PROGRAM....23

APPENDIX IV - TEST PROBLEMS....24

SUMMARY OF RESEARCH ACCOMPLISHMENTS

This research has developed a solution procedure for a practical, new combinatorial optimization problem, namely, problem (*) of page 4 of this technical report. The solution procedure also represents a new approach to combinatorial problems.

Problem (*) is a model for certain surveillance maintenance problems, and has not previously appeared in the literature of optimization or industrial engineering. Therefore, this research is bringing this important problem to the attention of researchers and specialists in the field of optimization and industrial engineering.

Our procedure for solving problem (*) is a heuristic that produces good quality, approximate solutions. This heuristic is designed to utilize solution guesses from the user, and incorporate new fast heuristics for the standard set-covering problem. Hence, this procedure has the potential to "evolve" and "develop", as more computational experiments are performed, and researchers become aware of problem (*).

This heuristic introduces a new approach to combinatorial problems. Combinations of sensor mechanisms are "lined up" as (binary coded) integers, and then a discrete line search (that is, a line search on integers only) is performed. Hence, a discrete optimization problem is transformed in a way that allows basic procedures of nonlinear programming to be applied. This should be a useful approach for a host of NP-complete combinatorial problems (the knapsack problem, for instance).

Finally, the cubical subdivision of the set S described in Appendix I of this report should have interesting implications for research on surveillance sensors.

SECTION 1

INTRODUCTION

This is the final report for research contract DASG60-90-C-0142 with the United States Army Strategic Defense Command, Huntsville, Alabama. This is also the third of three technical reports on the research contract. The first and second preliminary technical reports are included in this final report as Appendix I and Appendix II respectively.

The subject of this research contract is the problem labeled (*) below:

(*)Given: S, a subset of the 3-dimensional space, R^3; subsets S_{ij}, i=1,...,m, j=1,...,n, that cover S in the sense that S is the union of all the S_{ij}'s; two numbers t_{ij} and c_{ij} that will be interpreted as "time" and "cost" respectively. Required: To determine an optimal collection of the S_{ij}'s that cover S, with no more than one S_{ij} for each i, and optimality defined in terms of minimizing $\max\{t_{ij} : S_{ij}$ is in the collection$\}$ or minimizing the total cost of the S_{ij}'s in the collection.

This set-covering problem shall be referred to as problem (*). Problem (*) is a representation for a class of surveillance maintenance problems including the problem of restoring surveillance coverage for a system of surveillance sensor mechanisms after a sensor mechanism (in the system) fails. The set S represents the space under surveillance; each subset S_{ij} represents the portion of S covered by the i-th surveillance sensor mechanism functioning in its j-th optional deployment/positioning; each c_{ij} represents the cost of the j-th deployment of the i-th sensor mechanism; t_{ij} represents the time required for the i-th sensor mechanism to be placed in its j-th

optional deployment. Selecting a minimum-cost set of S_{ij}'s covering S corresponds to restoring surveillance coverage at minimum cost, whereas selecting a minimax-time set of S_{ij}'s covering S corresponds to restoring surveillance coverage within the shortest time possible.

For any procedure to solve problem (*), it is necessary to be able to determine whether a given selection of the S_{ij}'s define a cover for S. This question is the subject of our first preliminary technical report. In addressing this question, we find information about the sizes of potential intruders (into S) to be very helpful. This information allows us to avoid putting stringent conditions on S_{ij} or S. Section 2 of this report gives a summary of the preliminary technical report.

Also in section 2 of this report, our solution procedure for problem (*) is summarized. Appendix II contains a detailed description of this procedure. Partly because problem (*) is yet to appear in established literature, our procedure for solving problem (*) is not a variant of any existing procedures. Section 3 of this report evaluates certain computational features of this procedure.

Sections 3 and 4 of this report were not included in the preliminary technical reports. Using various test problems, section 3 contains information on the quality of solutions, and discusses some design choices. The computational complexity of the procedure is already estimated in Appendix II as $o(|S|m^2\log_{1.3}2^m)$.

Section 4 describes three extensions of problem (*) as possibilities for further work. These are resource-constrained extensions, commitment-constrained extensions, and multiple-layer coverage extensions. Such extensions incorporate a more realistic setting for problem (*).

To close this introduction, a clarification of one terminology that will be encountered

frequently in this report is in order. Since it is a heuristic, our solution procedure is application-oriented, and so is the description of it. Hence, sometimes problem (*) is referred to as the surveillance maintenance problem, even though problem (*) is actually an optimization model that has the surveillance maintenance problem as its motivation, and one of several possible applications.

SECTION 2

DESCRIPTION OF SOLUTION PROCEDURE

Our solution procedure has been formally described in our first and second preliminary technical reports, DASG60/90/1 and DASG60/90/2, which are included here as Appendix I and Appendix II respectively. In this section, we describe our motivation for the results presented in those preliminary technical reports. Informal summaries of the results are also given.

2.1 Determining Whether S is Covered

Any method for solving problem (*) must include a procedure for checking whether the set S is equal to the union of any selection of the S_{ij}'s, that is, whether S is covered by a given selection of the S_{ij}'s. If the set S were a finite set, then it would be a straightforward matter to do this checking on a digital computer. But, in problem (*), the set S in not finite; it is a 3-manifold in R^3 (that is, a full-dimensional volume in R^3). This task of checking whether S is covered is the subject of our first preliminary technical report.

In the case that S is not a finite set, it may be very difficult or even impossible to check whether S is covered. To illustrate this point, let us consider two special examples. First, suppose S and the S_{ij}'s are rectangles in R^2. Figure 2.1 below portrays a possible situation that may develop. This diagram suggests that the task (checking whether S is covered) is not impossible in this case, but could involve a great deal of record keeping.

Now suppose S is a rectangle in R^2, and some of the S_{ij}'s are intersections of circles and

S. Figure 2.2 below portrays a possible picture in this case. This diagram suggests that it may be impossible to check whether S is covered, unless further information is brought to bear on the case.

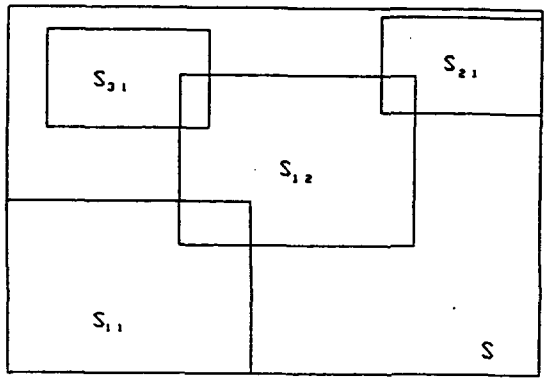


Figure 2.1

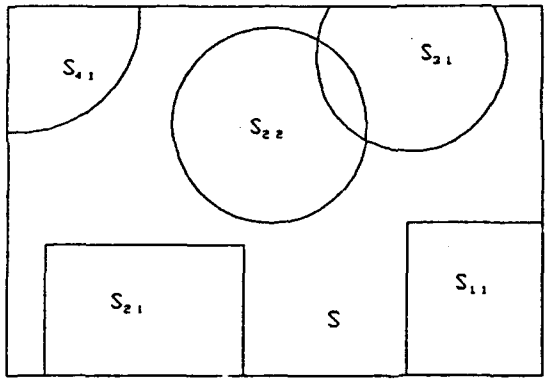


Figure 2.2

In problem (*), S corresponds to the space under surveillance, and each S_{ij} corresponds to the portion of S covered by the span of some sensor mechanism (in a specified deployment). S_{ij} depends on sensor technology, whereas S is a fixed volume. Since sensors have various non-surveillance uses in the real world, it may not be realistic to impose restrictions on the S_{ij}'s just to make it convenient to check whether S is covered.

Fortunately, the nature of surveillance maintenance itself provides a useful information, namely , the size of a potential intruder. Suppose x is the size of the smallest potential intruder into S. S may be "subdivided" into a finite number of pieces that fit together suitably, each piece of size at most x. Then the surveillance sensor mechanisms only have to cover (all of) the vertices (corners) of those pieces. Since there is a finite number of such pieces, this essentially reduces S to a finite set for the purpose of comparing S to the union any given selection of the S_{ij}'s.

In the first preliminary technical report, alternative ways of subdividing S for this purpose were considered, and a particular kind of subdivision, the cubical subdivision, was found to be the most suitable. The cubical subdivision allows an easy denumeration (line-up) of the pieces, and does not involve too many vertices. Other desirable properties of the cubical subdivision were also described in the technical report.

The literature of piecewise linear topology contains various types of such a subdivision of manifolds. Subdivisions of manifolds are also used extensively for homotopy methods in mathematical programming [1][19] and finite element methods [15][18].

2.2 Our Procedure For Problem (*)

It has been shown that, for the purpose of solving problem (*), S may be regarded as a finite set (by virtue of results given in Appendix I). But, even with S regarded as a finite set, problem (*) remains ordinarily harder than the standard set-covering problem, since problem (*) reduces to the standard set-covering only when n=1 and cost is the objective function. Therefore, one may not rely on standard set-covering procedures to solve problem (*). Our procedure for

solving problem (*) is the subject of the second preliminary technical report (Appendix II).

Problem (*) is an NP-complete combinatorial problem, as the standard set-covering is NP-complete. By the theory of algorithms [7][14], this means that it is highly unlikely that anyone can ever find an "efficient" procedure that computes an exact solution for problem (*). As an illustration of the meaning of this NP-completeness statement, solving an instance of problem (*), with 10 sensor mechanisms and 5 deployment options for each sensor mechanism, may take years on a fast digital computer doing one operation per micro-second, if an exact solution is required. Hence, a realistic approach for real world problems is to use procedures that produce good approximate solutions. Such approximate procedures are known as heuristic procedures or, simply, heuristics.

Since they are designed to produce approximate solutions, rather than exact solutions (even though they may generate exact solutions quite often), heuristics are usually designed to meet quality specifications derived from particular applications. Indeed, it is generally believed that any heuristic must be motivated from the needs of some specific applications.

Accordingly, our procedure for solving problem (*) is a heuristic whose features are motivated by the needs of surveillance maintenance applications. This heuristic is designed to have the following quality attributes: (i) ability to produce an approximate solution in good run time, (ii) ability to produce approximate solutions that are close to exact solutions, (iii) ability to utilize initial solution guesses given by the user or obtained from earlier iterations, (iv) ability to profitably incorporate new results on standard set-covering and related combinatorial problems.

This heuristic consists of the following main components: (i) a discrete line search (a line search wherein the function is evaluated at integral points only) that selects combinations of

sensor mechanisms, (ii) a random selection of deployments for combinations of sensor mechanisms, and (iii) a "greedy" procedure for computing cover value for any given deployment of sensor mechanisms.

The discrete line search does "skip and search" operations over the set of all possible combinations of sensor mechanisms. First, it goes through all k-subsets (of sensor mechanisms), then through all (k+1)-subsets, and so on. This search procedure allows the user to input a guess of how many sensor mechanisms may suffice to cover the space optimally, that is, the starting k, but this is not required. This guess is used to initiate the search, and naturally loses its effect if it is misleading.

If a selected deployment (of a given combination of sensor mechanisms) does not cover the space, the heuristic makes another selection of deployment. This way, each combination of sensor mechanism gets a number of "chances"; the number of chances given to each combination is determined at heuristic initialization.

Computational experiments indicate that the "greedy method" for computing cover value (cost or time) is a good choice. One can show that a related divide-and-conquer method suited to a parallel (processing) computer will perform even better than that greedy method.

Appendix II gives other details of this heuristic for problem (*). It also includes the source code of a computer program (in C) that implements the heuristic. This program is included as a subroutine in the larger computer program used for the computational experiments discussed in section 3 of this report.

SECTION 3

COMPUTATIONAL EXPERIMENTS

The computational experiments reported here serve two purposes. First, they are designed to demonstrate that our procedure for problem (*) generates solutions close to optimal solutions. Secondly, they are used to examine some of the design choices that define the procedure. Such a numerical evaluation of design choices may be necessary even if performance guarantees have already been given analytically.

3.1 Design Choices

The main design choices in our procedure for problem (*) are:

i. the use of binary code to "line-up" all combinations of sensor mechanisms,

ii. the restart feature, instead of a one-pass search,

iii. the use of random deployment, for any given combination of sensor mechanisms, instead of an orderly enumeration of possibilities, and

iv. the use of a greedy method for computing the objective function values (once the deployment is determined).

Alternatives to the binary code, such as the gray code, were considered before the binary code was chosen. Computational experiments (using a program written in BASIC with function plotting, etc.) put the binary code ahead of other options for two reasons. First, the binary code is easy to explain. Secondly, the function plots obtained for binary codes were more amenable

to function minimization than they are for other codes. To explain this apparent superiority of the binary code, we focused on k-terms (that is, terms of the binary code sequence that have exactly k 1's in their digits), and discovered some orderliness that led us to the "restart" idea.

To see this k-term property of binary code sequences, consider the k-terms of the following sequence.

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,

1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.

The sequences of k-terms are

0001, 0010, 0100, 1000,

0011, 0101, 0110, 1001, 1010, 1100,

0111, 1011, 1101, 1110,

1111.

These suggested to us that it might be advantageous to concentrate the line search on one sequence of k-terms at a time. Hence, the idea of restart. Computational experiments with test problems (described below) suggest that the "restart" feature is indeed a good idea, besides allowing the user to input a guess about how many sensor mechanisms may suffice to cover the space.

With regard to randomness in the deployment choice, the only alternative is to enumerate possibilities on the basis of information obtained from some pre-processing of data concerning deployment options. We did not perform computations with this alternative because it seems too much work for too little reward. The deployment choice aspect has significant scope for future work. However, the randomness seems to have no adverse effect on the results when relatively

few of the sensor mechanisms have multiple deployment options.

The use of a greedy method for computing objective function values is convenient on account of speed. Any sensible heuristic for the standard set-covering problem may do equally well. A "divide-and-conquer" alternative was considered, but this is more suited to a parallel (processing) computer.

3.2 Hardware and Programming

To some extent, all computational experiments reflect hardware and programming. Our computer program for implementing the procedure is written in C language so as to take advantage of C's efficacy in handling complicated data structures. We had presumed that subdivisions of the set S might require such data structures.

Development started with an IBM RT PC in a UNIX environment, but we switched to an IBM AT PC (286) and an IBM PS/2 Model 80, so as to take advantage of existing utility packages. The source code included here as an Appendix is in Microsoft C, using function prototyping and other features of ANSI C. Our test problems are of the size that will run with DOS. It is a simple matter to make the little adjustments (function declaration, etc.) that will make this code run in the environment of other operating systems.

Our program accepts inputs in form of files stored in "binary" (for space use efficiency). The user is required to supply the file name when the file is to be read or written into. The user also selects the objective function - cost or time. Next, the user is asked to guess (if possible) how many sensor mechanisms may suffice to cover the space.

3.3 Test Problems

Our test problems are not of the most general kind. They have been designed with the surveillance application in mind. We believe that the surveillance application will ordinarily have (i) many pieces in the cubical subdivision of S (Appendix I), (ii) few sensor mechanisms, (iii) most sensor mechanisms having exactly one deployment option. Test problems reflecting these observations, and computed solutions are given in Appendix IV.

Each problem has six sensor mechanisms, and each sensor mechanism has at most two deployment options. The coverage span of each sensor mechanism is assumed to be a sphere, and the set S is assumed to be a rectangular volume. In our experience, when computed "cost" solution is not optimal, the computed "time" solution usually gives help. In any case, supplying "guesses" always helps.

It is clear from an inspection of those test problems that there is need for more computational experiments, especially ones involving large-scale problems, the kind that may require a computing environment such as that provided by OS/2 or UNIX.

SECTION 4

DIRECTIONS FOR FUTURE WORK

Directions for further research related to problem (*) are suggested in this section. Three classes of extensions of problem (*) will be described. The first class has to do with making problem (*) more realistic in terms of resource constraints, for example, budget constraint and time constraint. The second class is about how existing commitments may constrain the choice of sensor mechanisms and their deployment options. The third class of extensions is concerned with multiple-layer surveillance coverage for selected portions of the space under surveillance. In addition, further computational experiments on our procedure for problem (*) will continue to be of interest.

4.1 Resource-Constrained Extensions

Each time an instance of problem (*) is solved, the objective function is either "cost" or "time". But, in the real world, one may be interested in both time and cost simultaneously. It may be desirable either to minimize surveillance rescheduling cost, with a limit on the time it takes to complete the rescheduling, or to minimize the time it takes to complete surveillance rescheduling, with a limit on the cost of doing the rescheduling.

The following is a formal statement of these two resource-constrained extensions of problem (*):

(1) Problem (*), with objective function replaced by minimize $\sum c_{ij,}$, subject to the

constraint that S_{ij} is in the collection, and $\max\{t_{ij}\} \leq T$

(2) Problem (*), with objective function replaced by minimize $\max\{t_{ij}\}$ subject to the constraint that S_{ij} is in the collection and $\sum c_{ij} \leq C$

Here, T is the time limit, and C is the cost limit. These resource-constrained problems are "mixed", because each problem is a mixture of combinatorial and continuous variable optimization problems.

4.2 Commitment-Constrained Extensions

In the real world, existing contracts, concerning orders and supplies, will ordinarily constrain new decisions. Therefore, decisions involved in rescheduling surveillance coverage may be subject to commitment constraints. Existing commitments on orders and supplies for surveillance maintenance may be reflected in problem (*) in form of pre-determined selection of some sensor mechanisms or some deployment options. Indeed, the standard set-covering problem may be regarded as an extreme-case member of this class of extensions of problem (*); it is the case where each surveillance sensor mechanism is allowed exactly one deployment option, an option probably fixed by some legal contract.

4.3 Multiple-Layer Surveillance Coverage

It is possible that some portions of the space under surveillance in problem (*) are so special that they require a multiple-layer surveillance coverage. For example, suppose it is necessary to ensure that portions of the boundaries of the space be covered by at least 2 sensor mechanisms. This may be required to satisfy some reliability requirements. We would refer to

this as a 2-layer coverage problem.

This gives rise to a k-layer extension of problem (*). Problem (*) is obviously a 1-layer coverage problem. Now, if we allow portions of the space to have 0-layer coverage, and other portions to have positive-layer coverage, then we have a very practical extension of problem (*).

REFERENCES

1. Awoniyi, S.A. & M.J. Todd "An Efficient Simplicial Algorithm For Computing A Zero Of A Convex Union Of Smooth Functions" Mathematical Programming 25 (1983) pp. 83-108.
2. Balas, E. "Cutting Planes From Conditional Bounds: A New Approach To Set Covering," Mathematical Programming Study 12 (1980) 19-36.
3. Balas, E. & A. Ho, "Set Covering Algorithms Using Cutting Planes, Heuristics and Subgradient Optimization: A Computational Study", Mathematical Programming Study 12, (1980) pp. 37-60.
4. Balas, E. & S.M.Ng., "On The Set Covering Polytope: II, Lifting The Facets With Coefficients in {0,1,2}" Mathematical Programming, Vol. 45, No. 1 (1989) pp. 1-20.
5. Christofides, N. and S. Korman, "A Computational Survey of Methods For The Set Covering Problem," Report 73/2, Imperial College of Science and Technology (London, 1973)
6. Cornuejols, G. and A. Sassano, "On The 0,1 Facets Of The Set Covering Polytope," Mathematical Programming, 43 (1989) 45-55.
7. French, S., Sequencing and Scheduling, Ellis Horwood (1982)
8. Fulkerson, D.R., G.L. Nemhauser and L.E. Trotter, "Two Computationally Difficult Set Covering Problems That Arise In Computing The 1-Width Of Incidence Matrices Of Steiner Triple Systems," Mathematical Progamming Study 2 (1974) 72-81.
9. Garfinkel, R.S. & G.L. Nemhauser, Integer Programming, John Wiley & Sons (1972).

10. Lemke, C.E., H.M. Salkin and K. Spielberg, "Set Covering By Single-Branch Enumeration With Linear Programming Subproblems," Operations Research 19 (1971) 998-1022.

11. Murty, K., Linear and Combinatorial Programming, John Wiley & Sons (1976).

12. Nobili, P. & A. Sassano, "Facets and Lifting Procedures For The Set Covering Polytope" Mathematical Programming, Vol. 45, No. 2 (1989) pp. 111-138.

13. O'hEigeartaigh, M., J.K. Lenstra & A.H.G. Rinnooy Kan (Editors), Combinatorial Optimization: Annotated Bibliographies, John Wiley & Sons (1985)

14. Parker, R.G. & R.L. Rardin, Discrete Optimization Academic Press (1988)

15. Reddy, J.N., An Introduction To The Finite Element Method, McGraw-Hill (1984)

16. Saaty, T.L., Optimization In Integers and Related Extremal Problems, McGraw-Hill (1970)

17. Sassano, A., "On The Facial Structure of The Set Covering Polytope", Mathematical Programming, Vol. 44, No. 1 (1989) pp. 181-202.

18. Tapley, B.D., (Editor) Eshbach's Handbook of Engineering Fundamentals, John Wiley (1990).

19. Todd, M.J., The Computation Of Fixed Points and Applications, Springer-Verlag, New York (1976).

20. Zemel, E., "Lifting The Facets of Zero-One Polytopes," Mathematical Programming 15 (1978) 268-277.

APPENDIX 1

FIRST PRELIMINARY TECHNICAL REPORT

DEPARTMENT OF INDUSTRIAL ENGINEERING

FAMU/FSU COLLEGE OF ENGINEERING

Research Progress Report DASG60/90/1

SUBDIVIDING 3-MANIFOLDS FOR SURVEILLANCE MAINTENANCE

by

Samuel Awoniyi, Ph.D.
&
Lester Frair, Ph.D.

November 1990

This is a preliminary technical report for
the USASDC research grant #DASG60-90-C-0142

SUBDIVIDING 3-MANIFOLDS FOR SURVEILLANCE MAINTENANCE

by

S. Awoniyi & L. Frair

Abstract

This report contains a procedure for checking whether a given set of surveillance sensor mechanisms is sufficient to cover a given airspace, say S, under surveillance. This procedure consists of using information about the size of the smallest potential intruder (into S) to subdivide S into pieces that fit together suitably. The vertices of these pieces constitute a finite set of points that the sensor mechanisms must cover. Hence, this procedure essentially replaces S with a finite set; then, checking whether S is covered by the sensor mechanisms becomes a straightforward task on a digital computer. Also included (as Appendix II) in this report is a menu-driven computer program (in C programming language) that implements that procedure; it subdivides S and denumerates (assigns numbers to) its pieces, in a manner that helps record-keeping. This computer program will be incorporated into the computer codes that will be delivered to the US Army Strategic Defense Command under the research contract DASG-60-90-C-0142. In the mean time, this computer program may be used for experimentation and demonstrations.

1. Objective of Results

The results given in this report address aspects of the generalized set-covering problem described in the research grant DASG-60-90-C-0142, sponsored by the United States Army Strategic Defense Command. The problem is recalled here:

> (*)Given: S, a subset of the 3-dimensional space, R^3; subsets S_{ij}, i=1..., m, j=1,...,n, that cover S in the sense that S is the union of all the S_{ij}'s; two numbers t_{ij} and c_{ij} that will be interpreted as "time" and "cost" respectively. Required: To determine an optimal collection of the S_{ij}'s that cover S, with no more than one S_{ij} for each i, and optimality defined in terms of minimizing max $\{t_{ij}:S_{ij}$ is in the collection$\}$ or minimizing the total costs of the S_{ij}'s in the collection.

This set-covering problem shall henceforth be referred to as problem (*).

Any method for solving problem (*) must include a procedure for checking whether the set S is equal to the union of some of the S_{ij}'s. If the set S were a finite set, then it would be a straightforward matter to do this checking on a digital computer.

In the surveillance applications of problem (*), the set S is not finite; S is a 3-dimensional manifold (that is, a full-dimensional volume in R^3). S represents the space under surveillance, whereas S_{ij} represents the portion of S that may be covered by the i-th surveillance sensor mechanism functioning in its j-th optional positioning.

If S is a 3-manifold without any special features, to determine whether S is equal to the union of some of the S_{ij}'s is a very hard task on a digital computer. But the surveillance maintenance situation allows us to regard S as "variably finite" (this term will be fully explained in Section 2). Utilizing this special feature of S, we give in this report a reasonable procedure

for comparing S and the union of any collection of the S_{ij}'s.

This procedure consists of (a) subdividing S into a finite number of pieces of suitable sizes, with each piece "contained" in at least one of the S_{ij}'s, and (b) denumerating (numbering) the pieces in a way that will facilitate efficient record-keeping. These results essentially replace S with a finite set $\{\sigma_1, \cdots \sigma_p\}$, where σ_k is the k-th piece of the subdivision, and p the total number of such pieces. For overall efficiency, it is desirable to make p as small as possible, while, at the same time, ensuring that it does not take much effort to check whether a piece is covered by an S_{ij}. Section 2 of this report gives details of these results. A computer code (in C language) implementing this procedure is included as Appendix II.

2. Description of Results

This section describes our procedure for comparing S and unions of the S_{ij}'s. We shall begin by making some problem-reducing observations on the surveillance maintenance situation. These observations are instrumental to our replacing S with a finite set, thereby making the set comparison task a straightforward one.

(*)Observations: (i) S is compact, since surveillance over an unbounded space is not practicable; (ii) surveillance sensor mechanisms do not have to cover every point of S; they only need to cover points in S that are not too far apart relative to the size of the smallest potential intruder into S.

To substantiate the claim contained in the second part of those observations, let us suppose S is a rectangular volume, and the smallest potential intruder into S is of size α. Let S be subdivided into polyhedral pieces (that is, pieces with straight boundaries) $\{\sigma_1,...,\sigma_P\}$, each σ_i with diameter less than α (the diameter of a set is the length of the longest straightline

contained in the set). The compactness of S guarantees the existence of such polyhedra pieces. Obviously, the surveillance sensor mechanisms only need to cover the vertices of the σ_i's, since that guarantees that any intruder will be detected. Figure 1 gives an illustration in R^2 of possible subdivisions of manifolds. Thus, S is effectively replaced with a finite set, the set of all vertices of the pieces of a subdivision of S. Hence, by virtue of the observations (*), S may be regarded as a "variably finite set". The qualifier "variably" is necessary because there are many different ways of subdividing S into pieces $\sigma_1,...,\sigma_p$, giving rise to many different finite sets for replacing S.

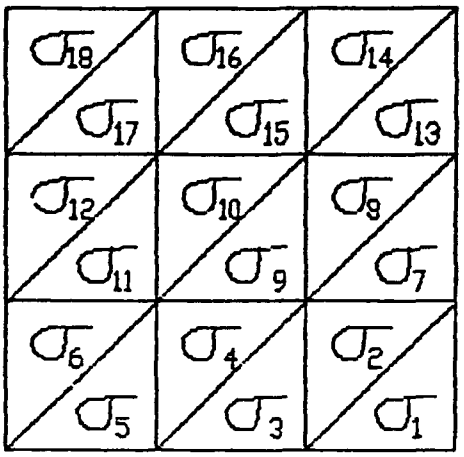
As one can see from Figure 1, some subdivisions of S are better than others. In choosing a subdivision of S, it is desirable to

(i) minimize the number of (subdivision) pieces; Figures 1(a) and 1(c) show examples of subdivisions with relatively too many pieces;

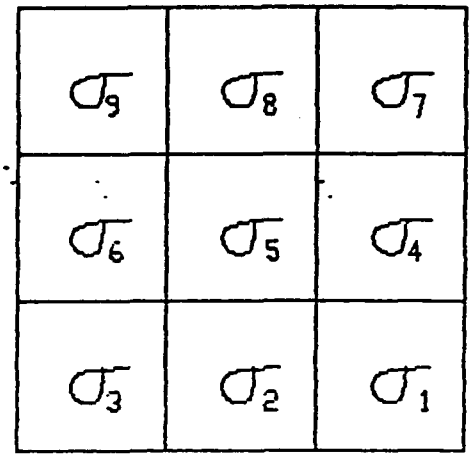
(ii) avoid repeating a vertex treatment in assigning (subdivision) pieces to the S_{ij}'s; such a repetition will occur if there are more pieces than vertices; Figure 1(a) illustrates this point;

(iii) ensure effective denumeration of pieces (effective in the sense that each piece number automatically identifies the corresponding piece, and vice versa), so as to facilitate efficient record keeping; in this regard, the subdivision of Figure 1(b) is superior to those of Figures 1(a) and 1(c).

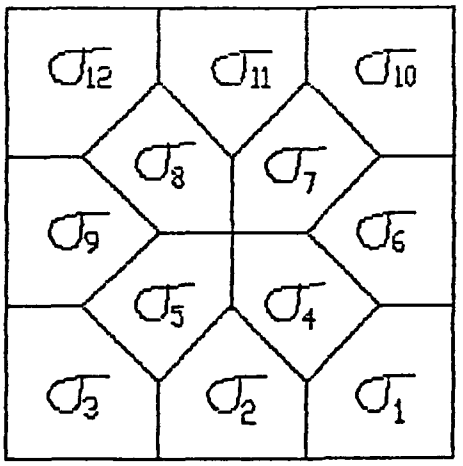
Evidently, the usual performance criteria for subdivisions in homotopy algorithms [5] , such as "directional density", do not apply here because subdivision traversing is not of interest in the context of surveillance maintenance. Appendix I gives a formalization of criteria (i) and (ii) above.



(a) Using 3-sided pieces



(b) Using 4-sided pieces



(c) Using 5-sided pieces

Figure 1: Illustration in R^2

On the basis of the three criteria listed above, the subdivision of Figure 1(b) is clearly superior to those of Figures 1(a) and 1(c). In fact, in R^2, when one considers using k-sided pieces, for k=3,4,..., it is not hard to see that the subdivision in Figure 1(b) is superior to all others. In R^3, analogues of the subdivision of Figure 1(c) fail our criteria because they involve too many vertices and are too complicated for effective denumeration; such subdivisions are never used in the literature of homotopy algorithms and finite element methods [2].

Therefore, the feasible options reduce to choosing between 3-dimensional analogues of the subdivision of Figure 1(a), which are known as "triangulations", and 3-dimensional analogues of the subdivision of Figure 1(b), which we shall term "cubical subdivisions". Cubical subdivisions cover R^3 with identical cubes or rectangular blocks, whereas triangulations cover R^3 with tetrahedra. Now, triangulations fail our criteria on account of having more pieces (called simplices) than vertices (so vertex treatment repetition will occur), and because they are difficult to denumerate effectively. In contrast, the cubical subdivision has relatively few pieces (always less than the number of vertices), and is amenable to easy, effective denumeration. Indeed, for the cubical subdivision, only one vertex of each piece (its origin) needs to be assigned to the S_{ij}'s, because the other vertices (of the piece) are origins for other pieces, unless the piece happens to lie on some of the boundary of S. Figure 2 illustrates this point about the cubical subdivision. Hence, we have a case for choosing the cubical subdivision for our purposes here.

But to conclude the case for the cubical subdivision, it is necessary to consider the amount of record-keeping that must be done at the boundary of S, taking into account the types of S that may arise in the surveillance maintenance situation (consideration of the interior of S having been subsumed in the foregoing comparison of subdivisions of S). Now, since, by definition, S represents an airspace, one may assume that S is essentially a 3-manifold of the type shown in

$S = 3 \times 4$ square; $\alpha = \sqrt{2}$

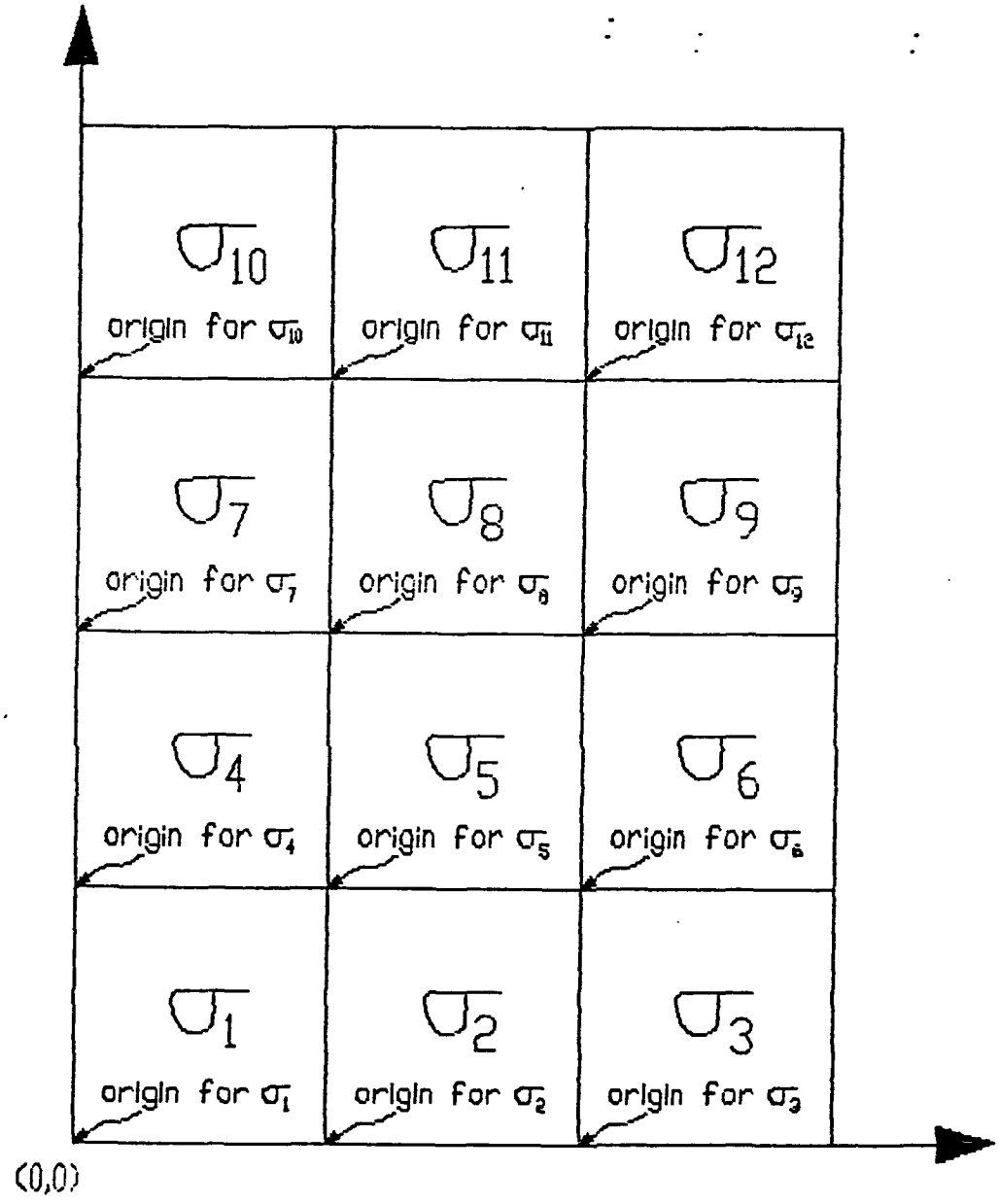


Figure 2: Efficacy of the Cubical Subdivision in R^2

Figure 3(a). As the manifold of Figure 3(a) is clearly homeomorphic to that in Figure 3(b), in a topological sense, one may then assume that S is essentially a rectangular volume, without any loss of generality. Hence, S indeed possesses the features that have been demonstrated (in the foregoing) as reinforcing the cubical subdivision's superiority. This concludes our case for choosing the cubical subdivision.

The foregoing analysis is the basis for a computer program that is included here as an appendix. This computer program gives a cubical subdivision of S, and denumerates its pieces. It will be included as a component of the final computer implementation of our solution of problem (*). Presently, this computer program may be used in computational experiments and demonstrations on the cubical subdivision.

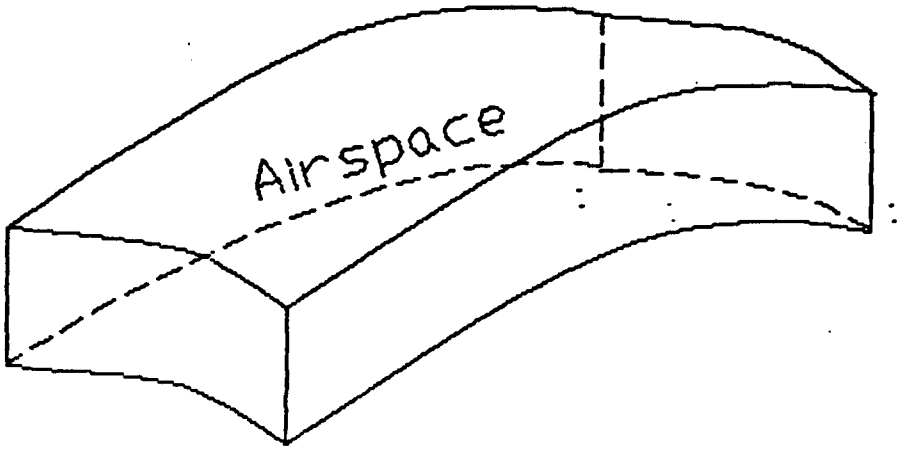
The computer program is menu-driven and user-friendly. First, it requires the user to supply the dimensions (length, breadth, height) of the rectangular volume S. Then the user makes a selection from the following menu:

A - To Compute Number Assigned to a Piece

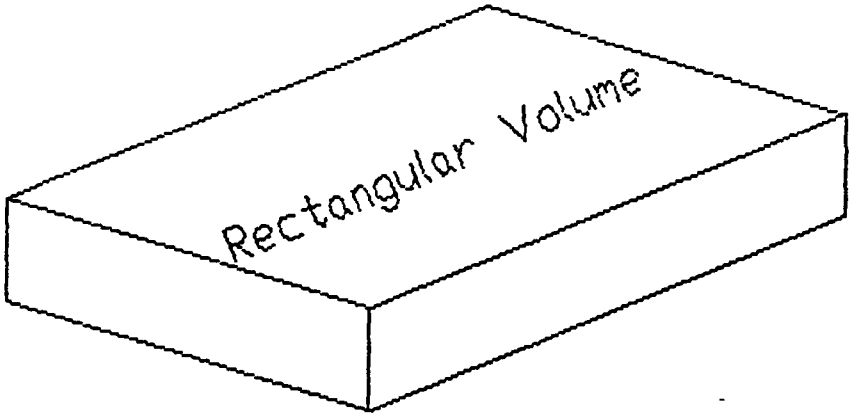
B - To Compute the Piece for a Given Number

C - To Return To Operation System

If A is selected, then the user is asked to supply the coordinates of the piece's origin; the computer program then returns the piece's number. If B is selected, then the user is asked to give the number; the program then returns the origin of the corresponding piece, unless the given number is too big to be a piece number. In the case that the number is too big, the program returns an error message.



3 (a) S = Airspace



3 (b) S After Homeomorphic Transformation

Figure 3: The Form of S In Problem (*)

3. Concluding Remarks

Besides allowing us to compare S to collections of the S_{ij}'s, replacing S with a finite set also transforms problem (*) into a generalized discrete set-covering problem that is amenable to a host of discrete optimization strategies and heuristics. These strategies include divide-and-conquer, discrete line search, decomposition, and enumeration.

To employ these strategies to solve problem (*), we shall use the computer program included here as a "function" (or subroutine). This function will be called for doing "initial set-up" (implicitly replacing S and the S_{ij}'s with finite sets), and for checking for feasibility when collections of the S_{ij}'s are examined.

Since each S_{ij} will contain relatively few of the pieces (of the cubical subdivision of S), S_{ij} may be represented as a "linked list". The C programming language is ideal for such a data structure.

Now, our analysis (in Section 2) that resulted in the choice of the cubical subdivision is relatively informal. A more formal analysis resulting in the same conclusion is included here as Appendix I.

REFERENCES

[1] Parker, R.G. & R.L. Rardin, Discrete Optimization, Academic Press (1988)

[2] Reddy, J.N., An Introduction to the Finite Element Method, McGraw-Hill (1984)

[3] Saaty, T.L., Optimization in Integers and Related Extremal Problems, McGraw-Hill (1970)

[4] Tapley B.D., (Editor) Eshbach's Handbook of Engineering Fundamentals, John Wiley (1990)

[5] Todd, M.J., The Computation of Fixed Points and Applications, Springer-Verlag, New York (1976)

APPENDIX I

A FORMAL CASE FOR THE CUBICAL SUBDIVISION

Here, we present in R^2 a more rigorous version of our case for the cubical subdivision. We shall also indicate how this analysis may be extended to R^3. First, a number of concepts will be defined.

Definition: For any polygon, σ, in R^2, the length of the longest side shall be called the polygon's surveillance diameter, and will be denoted by sdiam(σ).

The number sdiam(σ) is the upper bound on the size of intruders that may enter $\hat{\sigma}$ undetected by surveillance sensors that cover the vertices of σ only. Our next definition is a performance measure (for subdivisions) summarizing two of the three criteria given in Section 2.

Definition: For any n-sided polygon, σ, in R^2 with area A, the number A/n shall be called the polygon's index of surveillance subdivision efficacy, and will be denoted by isse(σ).

If, for a polygon σ, sdiam(σ) is acceptably small and isse(σ) is relatively large, then σ is a good candidate for use as a subdivision's basic building block.

The following lemma follows easily from the definitions above:

Lemma: (i) If σ is an equilateral triangle with side α, then

$$\text{sdiam}(\sigma) = \alpha \text{ and } \text{isse}(\sigma) = \alpha^2/4\sqrt{3}$$

(ii) If σ is a square of side length α, then

$$\text{sdiam}(\sigma) = \alpha \text{ and } \text{isse}(\sigma) = \alpha^2/4$$

Hence, for surveillance purposes in R^2, it is better to subdivide S into squares rather than equilateral triangles, especially if S happens to be a rectangular area. However, that does not yet indicate that squares are preferable to all triangles; this is the object of the following proposition.

Proposition: For any triangle, σ, in R^2 with sdiam(σ) = α ,

$$\text{isse}(\sigma) \leq \alpha^2/4\sqrt{3}$$

Proof (outline): It suffices to show that the area of the triangle σ does not exceed $\sqrt{3}\alpha^2/4$ (the area of corresponding equilateral triangle). Since sdiam(σ) = α, the longest side of σ is of length α. With a simple construction, one can enlarge σ into an isosceles triangle with two sides equal to α.

Then, using an elementary optimization technique, one can show that if σ is isosceles with sdiam(σ) = α, then the area of σ does not exceed that of an equilateral triangle with side length equal to α.

Thus, we have formally demonstrated that the cubical subdivision of S is superior to triangulations of S in R^2, even without using the fact that S is a rectangular area. This conclusion also holds true in R^3, but the analysis is a lot longer. In R^3, the tetrahedron replaces the triangle of R^2, and the cube replaces the square. Of course, the definitions of sdiam(σ) and isse (σ) must be recast -- "facet" replacing "line", "volume" replacing "area", etc.

Obviously, we have not considered k-sided polygons in R^2, for k=5,6,... This is because such polygons are too complicated (when interpreted for R^3) to allow sensible denumeration of subdivision pieces. They also fit poorly on the boundary of S, especially when S is a rectangular volume.

A P P E N D I X I I


```
* CUBICAL SUBDIVISION OF 3-MANIFOLDS FOR SURVEILLANCE MAINTENANCE: */
* WE ASSUME, WITHOUT LOSS OF GENERALITY, THAT THE MANIFOLD IS A    */
* RECTANGULAR 3-DIMENSIONAL VOLUME; THE MANIFOLD IS SUBDIVIDED INTO*/
* UNIT CUBES, AND THE CUBES ARE DENUMERATED FOR EFFICIENT RECORD   */
* KEEPING; THE USER SHALL SUPPLY THE MANIFOLD'S DIMENSIONS (LENGTH,*/
* BREADTH, AND HEIGHT), AND THEREAFTER SELECT FROM A MENU THAT     */
* GIVES A CUBE'S NUMBER WHEN THE CUBE'S ORIGIN COORDINATES ARE     */
* SUPPLIED, AND GIVES A CUBE'S ORIGIN COORDINATES WHEN THE CUBE'S  */
* NUMBER IS SUPPLIED.                                             */

oid pisnum(void);
oid numpis(void);
nt ll, bb, hh;
ain()
{
 char ss;
 printf("   THIS PROGRAM GIVES A CUBICAL SUBDIVISION OF A\n ");
 printf("         3-DIMENSIONAL RECTANGULAR BLOCK\n\n     ");
 printf("Please ENTER the dimensions of the block, p:q:r --> ");
 scanf("%d:%d:%d", &ll, &bb, &hh);
 start:
 printf("\n\n");
 printf("\n       ENTER \n");
 printf("\n A -- To Compute Number Assigned to a Piece\n");
 printf("\n B -- To Compute the Piece for a Given Number\n");
 printf("\n C -- To Return to DOS\n");
 printf("\n       YOUR SELECTION ---> ");
 ss = getche(); printf("\n\n");
 switch(ss)
        {
        case 'A':
          pisnum(); break;
        case 'a':
          pisnum(); break;
        case 'B':
          numpis(); break;
        case 'b':
          numpis(); break;
        case 'C':
          exit(0);
        case 'c':
          exit(0);
        default:
          goto start;
        }
 printf("\n\n");
 printf("-------------------------------------------------\n\n\n");
 printf("Press any key to continue");
 ss = getch();
 goto start;
 }
```

```
/* THIS PROCEDURE COMPUTES THE NUMBER ASSIGNED TO ANY */
/* PIECE OF THE CUBIC SUBDIVISION OF THE BLOCK. YOU ARE */
/* REQUIRED TO INPUT THE ORIGIN OF THE PIECE.\n\n*/

void pisnum(void)
{
  int xx, yy, zz, temp;
  start1:
  printf("Please ENTER the origin coordinates, x:y:z --> ");
  scanf("%d:%d:%d", &xx, &yy, &zz);
  if(xx>ll || yy>bb || zz>hh)
    {printf("\n\n!^!^!^!^!^!^!^INPUT ERROR! TRY AGAIN PLEASE\n\n\n");
     goto start1; }
  temp = (ll*bb*zz)+(ll*yy)+xx+1;
  printf("\nFor the piece with origin (%d,%d,%d), ", xx, yy, zz);
  printf("the corresponding number is %4d \n", temp);
}


/* THIS PROCEDURE COMPUTES THE ORIGIN OF THE  PIECE */
/* CORRESPONDING TO ANY GIVEN NUMBER*/

void numpis(void)
{
  int num, xx, yy, zz, temp;
  start2:
  printf("Please ENTER the number --> ");
  scanf("%d", &num);  printf("\n");
  if (num>(ll*bb*hh))
       {printf("\n\n!^!^!^!^!^!^!^INPUT ERROR! TRY AGAIN PLEASE\n\n");
       goto start2;
       }
  num-=1;   temp = ll*bb;
  zz = ( num - (num%temp) )/temp;  num -= zz*temp;
  yy = ( num - (num%ll) )/ll;
  xx = num - yy*ll;
  printf("The corresponding piece has (%d,%d,%d)  ", xx, yy, zz);
  printf("as origin coordinates\n\n");
}
```

APPENDIX II

SECOND PRELIMINARY TECHNICAL REPORT

DEPARTMENT OF INDUSTRIAL ENGINEERING

FAMU/FSU COLLEGE OF ENGINEERING

Research Progress Report DASG60/90/2

A RESTART PROCEDURE FOR A GENERALIZED SET-COVERING PROBLEM IN SURVEILLANCE MAINTENANCE

by

Samuel Awoniyi
&
Les Frair

February 1991

This is a preliminary technical report for
the USASDC research grant #DASG-90-C-0142

A RESTART PROCEDURE FOR A GENERALIZED SET-COVERING PROBLEM IN SURVEILLANCE MAINTENANCE

by

Samuel Awoniyi & Lester Frair

Abstract: We describe a heuristic procedure for the generalized set-covering problem defined in our research contract DASG 60-90-C-0142 with the United States Army Strategic Defense Command, Huntsville, Alabama. This heuristic is not a variant of any existing procedure, as the problem itself has not appeared in the literature. The quality attributes of this heuristic include (i) ability to produce an approximate solution in good run time, (ii) ability to produce approximate solutions that are close to exact solutions, (iii) ability to improve upon initial solution guesses given by the user or obtained from earlier iterations, (iv) ability to profitably incorporate new results on set-covering and related combinatorial problems. The main components of this heuristic are a discrete line search with restart, a random selection operation, and a greedy procedure on a subproblem. A computer program (in C language) implementing this heuristic is included as an Appendix. The next report on this research contract (which is also the final report) will give a full account of our computational experiments with this heuristic.

1. Introduction

This is the second of three technical reports fulfilling a requirement of our research contract DASG 60-90-C-0142 with the US Army Strategic Defense Command, Huntsville, Alabama. The goal of this research contract is to design solution procedures for the generalized set-covering problem described in (*) below.

(*) Given: S, a subset of the 3-dimensional space, R^3; subsets S_{ij}, i=1,...,m, j=1,...,n, that cover S in the sense that S is the union of all the S_{ij}'s; two numbers t_{ij} and c_{ij} that will be interpreted as "time" and "cost" respectively.

Required: To determine an optimal collection of the S_{ij}'s that cover S, with no more than one S_{ij} for each i, and optimality defined in terms of minimizing $\max\{t_{ij} : S_{ij} \text{ is in the collection}\}$ or minimizing the total cost of the S_{ij}'s in the collection.

This set-covering problem shall be referred to as problem (*). Problem (*) is a representation for a class of surveillance maintenance problems including the problem of restoring surveillance coverage for a system of surveillance sensor mechanisms after a sensor mechanism (in the system) fails. The set S represents the space under surveillance; each subset S_{ij} represents the portion of S covered by the i-th surveillance sensor mechanism functioning in its j-th optional deployment/positioning; each c_{ij} represents the cost of the j-th deployment of the i-th sensor mechanism; t_{ij} represents the time required for the i-th

sensor mechanism to be placed in its j-th optional deployment. Selecting a minimum-cost set of S_{ij}'s covering S corresponds to restoring surveillance coverage at minimum cost, whereas selecting a minimax-time set of S_{ij}'s covering S corresponds to restoring surveillance coverage within the shortest time possible.

Our first technical report is entitled "Subdividing 3-Manifolds For Surveillance Maintenance", and our third technical report shall consist of a summary together with a user-friendly computer implementation of the results presented in the first two technical reports. The third report will also give indications of further work that should be done in order to profitably adopt the results of these reports in the real world.

This second technical report describes a practical solution procedure for problem (*). To be able to see the origins of certain features of this procedure, it is necessary to understand the combinatorial nature of problem (*). Suppose there are d_i deployment possibilities for the i-th sensor mechanism. Then, there are $(d_1+1)(d_2+1)\ldots(d_m+1)$ valid combinations of the S_{ij}'s that must be considered explicitly or implicitly by any procedure that computes an <u>exact solution</u> to problem (*). If $d_i=n$, each i, then the number of combinations of the S_{ij}'s is $(n+1)^m$.

Suppose, also, that p computational steps are required to compute the (objective function) value of each of those combinations of S_{ij}'s. Then, the maximum number of computational steps needed to compute an exact solution to problem (*) is $(d_1+1)(d_2+1)\ldots(d_m+1)p$. Since we are assuming that it does not

depend on the S_{ij}'s, the number p ordinarily depends on the size of the space S; therefore, we shall henceforth write it as p(|S|).

We shall refer to the number $(d_1+1)(d_2+1)\ldots(d_m+1)$ p(|S|) as the computational upper bound (c.u.b.) for problem (*). The c.u.b. has some practical implication. To explain this practical implication of the c.u.b., let us consider a well-known special case of problem (*), namely the standard set-covering problem. When $d_i=1$, all i, (that is, each sensor mechanism has exactly one deployment option), problem (*) reduces to the standard set-covering problem (by virtue of results presented in our first technical report), and the c.u.b. becomes 2^mp(|S|). For instance, for p(|S|) > 1 and m = 50, the c.u.b. for the set-covering problem is at least 2^{50}, and this number of computational steps will take a fast digital computer doing one operation per micro-second about 35 years to complete (see pp 141 of [3] for more on computational speeds and their computer time requirements).

One clear conclusion from the foregoing observations concerning c.u.b. is that if a procedure must compute an exact solution for an instance of problem (*), then, unless m is suitably small, that procedure must keep p(|S|) small enough and avoid explicit consideration of most of the combinations of the S_{ij}'s. Now, this has a bearing on the theory of algorithms. It is well known in the theory of algorithms that the set-covering problem is NP-complete [1][7]; this means that it is highly unlikely that one can ever find a practical solution procedure (for the set-covering problem) that keeps p(|S|) small enough and, at the same time,

avoids explicit consideration of a good number of those combinations of the S_{ij}'s. Our third technical report expatiates further on this.

Hence, in solving problem (*) in the real world, unless S and m are suitably small, approximate solutions should be the goal; seeking an exact solution might take too long. Such approximate solutions should not be arbitrary, but must satisfy sensible specifications depending on the application. In section 2 of this report, we describe attributes that we consider desirable for an approximate solution procedure for problem (*), in view of the applications described above. We shall follow standard terminology, and refer to approximate solution procedures as heuristic procedures or heuristics.

In section 3, our heuristic procedure for problem (*) is described. Section 4 describes our computer program (in C) for implementing this heuristic, and the program's source code is included as an Appendix. Our next technical report will give a complete account of our computational experiments.

2. Quality Attributes of Heuristics for Problem (*)

Since heuristics are, by definition, designed to produce approximate solutions, rather than exact solutions, it is necessary to require of a heuristic certain attributes that constitute what may be termed "good quality". (Of course, a heuristic may very well produce exact solutions most of the times). For any heuristic, desirable quality attributes would ordinarily be

determined by the circumstances of particular real world applications. Hence, for the same problem model, different heuristics would be suited to different real world applications.

In this report, we shall regard problem (*) as having two broad application types, namely, the surveillance maintenance application with cost minimization as objective, and the surveillance maintenance application with maximum-time minimization as objective. It turns out that these two application types require slightly different heuristic attributes.

For the cost minimization version of problem (*), it is desirable that a heuristic be able to produce approximate solutions frequently close to exact solutions. The heuristic's speed is of lesser significance than the quality of approximate solutions generated. However, if a heuristic has a way of improving upon initial solution guesses, then good speed would ordinarily translate into approximate solution of good quality after repeated iterations.

For the maximum-time minimization version of problem (*), it may be desirable that a heuristic be fast, while at the same time producing an approximate solution of good quality. (Recall that the maximum-time minimization version is a representation for the problem of restoring surveillance cover as quickly as possible). As in the case of cost minimization, if a heuristic can produce an approximate solution that improves upon an initial solution guess, then good speed should result in a good final approximate solution. Good speed together with quality solutions would enhance the

usefulness of the heuristic, especially in emergency situations.

In general, a heuristic for problem (*), irrespective of type of objective function, should be designed to be able to incorporate future developments on set-covering and related combinatorial optimization problems. This is an especially desirable heuristic attribute because problem (*) is practical, NP-complete, new (in the literature), and likely to generate significant, new research activities in the near future.

Finally, as the question of surveillance maintenance concerns practical system security, usually involving the protection of precious human lives and property, it is desirable that a heuristic for problem (*) be capable of being described in terms that may be easily comprehended by persons who have day-to-day responsibility for such surveillance systems. Such clarity should enhance the heuristic's chances of being adopted in real-world situations.

3. Our Heuristic For Problem (*)

We describe in this section our heuristic for problem (*). This heuristic is designed to have the attributes described in section 2. In summary, the attributes are: (i) ability to produce an approximate solution in good run time (that is, good computational complexity), (ii) ability to produce approximate solutions that are close enough to exact solutions, (iii) ability to improve upon initial solution guesses given by the user or obtained from earlier iterations, (iv) ability to profitably incorporate new results on set-covering and related

combinatorial problems, (v) ease of comprehension, so as to secure user's trust.

Current literature contains nothing about problem (*) in its general form. Results on the set-covering problem [4] [7] are all that exist in current literature relating directly to problem (*). Therefore, our heuristic for problem (*) is not a variant of any existing procedure. The following is an informal, application-oriented description of the heuristic.

Heuristic

Step 1: Systematically skip and search through (a denumeration of) subsets of the given set of sensor mechanisms so as to find a combination of sensor mechanisms possibly "better" than the current combination, if any. Thereafter, go to step 2.

Step 2: Select a suitable deployment for the sensor mechanisms contained in the combination obtained from step 1 above. Thereafter, go to step 3.

Step 3: Compute the (objective function) value for the deployment of step 2. If that value is infinity (which means that the deployment does not cover the space), then go back to step 2 for another deployment, unless "chances" are exhausted. (The number of "chances" is set at heuristic initialization). Thereafter, go to step 4.

Step 4: Do updating and book-keeping to take into account the value obtained in step 3. Check stopping condition. If stopping condition is satisfied, then STOP; otherwise, go back to step 1,

with information to guide the next "skip and search".

The description above gets formalized as the underlined basic operations are explained in details. We will now describe each of these operations.

(a) "Systematically Skip and Search"

This is essentially a line search, except that the set over which this is done is the discrete set $\{1,\ldots,2^m-1\}$ coded in binary numbers. This set is constructed as follows. First, all combinations of the given sensor mechanisms (m of them) are "lined up" (denumerated) using a binary coding. For example, let us suppose m=3. Then the binary code line-up of all combinations of the three sensor mechanisms is given by the following:

001	010	011	100	101	110	111
{1}	{2}	{1,2}	{3}	{1,3}	{2,3}	{1,2,3}

Note that the binary numbers above form an increasing sequence for which the n-th term is obtained by adding 1 to the (n-1)-th term; in decimal numbers, that sequence is 1, 2, ... 7.

The rule for assigning binary numbers to combinations of sensor mechanisms is as follows:

Rule 3.1: Suppose a combination of sensor mechanisms is given. For i=1...m, if the i-th sensor mechanism is included in the given combination, then, counting from right to left, set the i-th digit (of the binary number) to 1; otherwise set it to 0. (The inverse of this operation is obvious).

Using Rule 3.1 above, all combinations of sensor mechanisms are

lined up in binary codes between 1 and 2^n-1. Hence a search through all combinations of sensor mechanisms becomes a search over the set $\{1,\ldots,2^n-1\}$.

But this "skip and search" does not cover all of $\{1,\ldots,2^n-1\}$ every time. The search confines itself to k-subsets (of the set of sensor mechanisms) only. After completing the search over k-subsets, it goes on to (k+1)-subsets, and thereafter goes on to (k+2)-subsets, and so on. At the initialization of the heuristic, the user is asked to provide a starting k, if he can; otherwise m/2 or (m+1)/2 is used as initial k. This initial k is the means by which the heuristic accepts initial solution guesses from the user.

Details of how this discrete line search starts off, and how it skips around (inside the set of binary coded integers described above) to find the next subset of sensor mechanisms will be given later in this section. For now, we assume that it does find the next subset, that is, a combination of sensor mechanisms.

(b) "Select a Suitable Deployment"

This is a random selection. Suppose d_i is the total number of deployment options for the i-th sensor mechanism. A random positive integral number, say r, is generated, and then deployment option $(r \bmod d_i)+1$ is chosen for the i-th sensor mechanism.

This way of selecting deployments has performed fairly well in our computational experiments. This aspect of our heuristic is amenable to certain advanced results in probability and measure theory, but that is not of much relevance to our immediate purpose here.

(c) "Compute the Objective Function Value"

This reduces to solving a standard set-covering problem. Any reasonable heuristic for the standard set-covering problem could be used here. We chose a greedy heuristic, but we know that a divide-and-conquer method should be preferable on parallel computers. Again, computational experiments indicate that this is a good choice.

Now, the given deployment of sensor mechanisms may not give a cover for the space. If the given deployment covers the space, then a finite value is returned by the greedy heuristic; otherwise the value of this combination is set to infinity. In the case that the given deployment does not give a cover, the corresponding combination of sensor mechanisms may get another chance. The deployment generation process described above may be repeated for the combination, unless the number of "chances" for the combination has reduced to 0. The number of chances is determined at heuristic initialization; it is the maximum number of random deployments that may be generated for each combination of sensor mechanisms each time it (the combination) is presented by the "skip and search" step. For every deployment generated, the number of chances reduces by 1.

(d) "Do Updating and Book-keeping"

First, some terminology. As the integers 1, 2, ... , 2^n-1, are points of the real line, we shall refer to their binary equivalents 0...01, 0...10, . . . , 1...11, as "points". Recall that each of these points corresponds to a subset of the given set of sensor

mechanisms. For a point p, the objective function value shall be denoted by f(p).

Our heuristic always maintains a 3-point v-pattern -- points p_L, p_M, p_R such that $p_L < p_M < p_R$ and $f(p_L) > f(p_M) < f(p_R)$. Figure 3.1 illustrates the 3-point v-pattern. The combination of sensor mechanisms from "skip and search" step corresponds to some point, say p_N, inside the interval $[p_L, p_R]$. "Updating" consists of

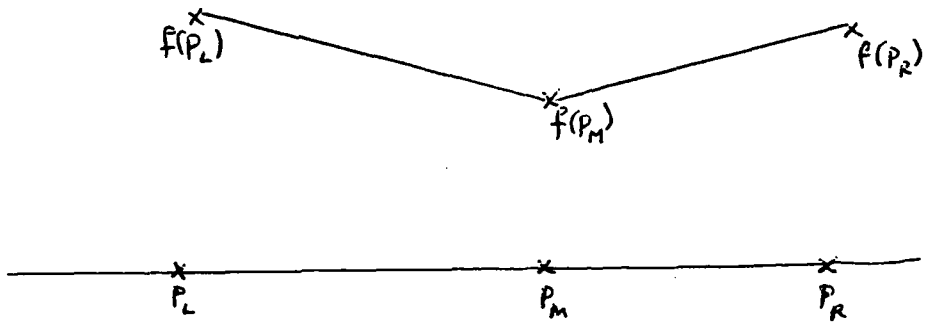


Figure 3.1 -- 3-point v-pattern

replacing either p_L or p_R with p_N in the 3-point v-pattern. Which one gets displaced depends on the magnitude of p_N relative to that of p_M and the magnitude of $f(p_N)$ relative to that of $f(p_M)$. If either $(p_N < p_M \ \& \ f(p_N) < f(p_M))$ or $(p_N > p_M \ \& \ f(p_N) > f(p_M))$, then p_R is displaced; otherwise p_L is displaced.

"Book-keeping" consists of recording the points constituting the current 3-point v-pattern and their objective function values. It also includes recording information about the "best" point so far; this is necessitated by the heuristic's need to start off and "restart" in a certain manner (see (f) below). As it goes from k-subsets to (k+1)-subsets, the heuristic uses a restart similar to

the first heuristic initialization (see (f) below).

(e) "Check Stopping Condition"

Assume the last "skip and search" step consists of searching among k-subsets. If k=m, our heuristic computes the objective function value for the point (1,...,1), prints all results, and STOPS. Suppose k < m. If current p_M is the only k-subset point in the interval $[p_L, p_R]$, then a switch to (k+1)-subsets is indicated as the heuristic returns to step 1; otherwise the heuristic returns to step 1 with information that the search is still on k-subsets.

(f) More details on the Line Search

Let L(k) denote the binary number with 1's as the first k digits and 0's as the last m-k digits, counting from right to left, and R(k) the binary number with 0's as the first m-k digits and 1's as the last k digits. Lemma 3.1 below indicates how we will use L(k) and R(k).

Definition For the binary code sequence

0...01, 0...10, 0...11, . . . , 1...11,

a term shall be called a k-term if it has exactly k 1's in its digits.

Lemma 3.1 All k-terms are contained in the interval [L(k),R(k)].

Proof: This becomes obvious when one remembers that the n-th term is obtained from the (n-1)-th term by adding 1. ##

To start searching through k-subsets, our heuristic sets p_L=L(k), p_R=R(k), and p_M equal to a k-term close to (L(k)+R(k))/2. Lemma 3.2 below shows how p_M may be obtained.

Lemma 3.2 (Skipping Lemma) Suppose k < m.

(i) Let p(k+1) be a (k+1)-term. Obtain a k-term q(k) from p(k+1) by setting to 0 the rightmost nonzero digit of p(k+1). Then, there is no other k-term between q(k) and p(k+1).

(ii) Let u(k) be a k-term. Obtain a (k+1)-term w(k+1) from u(k) by setting to 1 the rightmost zero digit of u(k). Then, there is no other (k+1)-term between u(k) and w(k+1).

Proof: Express terms in decimal numbers, and the desired conclusions follow from straightforward comparisons. ##

When it switches from k-subsets to (k+1)-subsets, the heuristic restarts the search the same way it starts the search through k-subsets, except that k is replaced by k+1.

When the heuristic starts or restarts, the objective function values may not give the desired v-pattern. This apparent difficulty is overcome by saving the best of the objective function values, if it is good enough (see "Book-keeping" under (d)), and thereafter setting $f(p_L)$ and $f(p_R)$ artificially so as to obtain a v-pattern.

After its initialization, the heuristic applies lemma 3.2 to the integral part of $(p_L+p_M)/2$ or $(p_M+p_R)/2$ to obtain p_N. It is this application of lemma 3.2 that constitutes "skipping" in this heuristic. The kind of skipping done here is similar to what might be done by a procedure doing discrete line search on the set of integers divisible by 7, then on the set of integers divisible by 9, thereafter on the set of integers divisible by 11, and so on.

To conclude this description of our heuristic, we now state its computational complexity (that is, its speed). First, our earlier

description of "Updating" needs a little adjustment so as to ensure a certain speed-up. Recall that p_N is obtained from $(p_L+p_M)/2$ or $(p_M+p_R)/2$, and then a new v-pattern is formed. Let us refer to the integral part of $(p_L+p_M)/2$ or $(p_M+p_R)/2$ as the "source" of p_N. To ensure that it discards about 1/4 of current interval with every "skip and search" step, the heuristic must replace p_N with its "source" as it (that is, p_N) is about to become a part of the 3-point pattern, and regard $f(p_N)$ as the function value for this "source". This adjustment ensures that the heuristic runs in $o(|S|m^2\log_{1.3}2_m)$ time, since our greedy procedure for set-covering runs in $o(|S|m)$ time. Here, $|S|$ is the total number of vertices of the cubical subdivision of S given in our first technical report.

4. Corresponding Computer Program

The major subroutines of our computer program are close to the heuristic operations described in section 2. The flow charts of Figures 4.1 and 4.2 display the logic of this computer program, and the following table describes a useful correspondence between the heuristic operations and the program subroutines.

Two simplifying assumptions have been made. First, we assume that the space S is a rectangular volume in R^3. We also assume that each surveillance sensor mechanism has a spherical coverage or span. These two assumptions are essentially assumptions about sensor technologies, and could be discarded without adversely affecting the basic features of the computer program. Such technological assumptions will be thoroughly considered in our

follow-up research work.

Heuristic Operation		Program Subroutine
		setup()
skip and search	<----->	restart()
		nextpt()
compute objective function	<----->	ptvalue()
updating and book-keeping	<----->	update()

Computational experiments with this program have been encouraging. The results of these computational experiments shall be fully described in our next technical report.

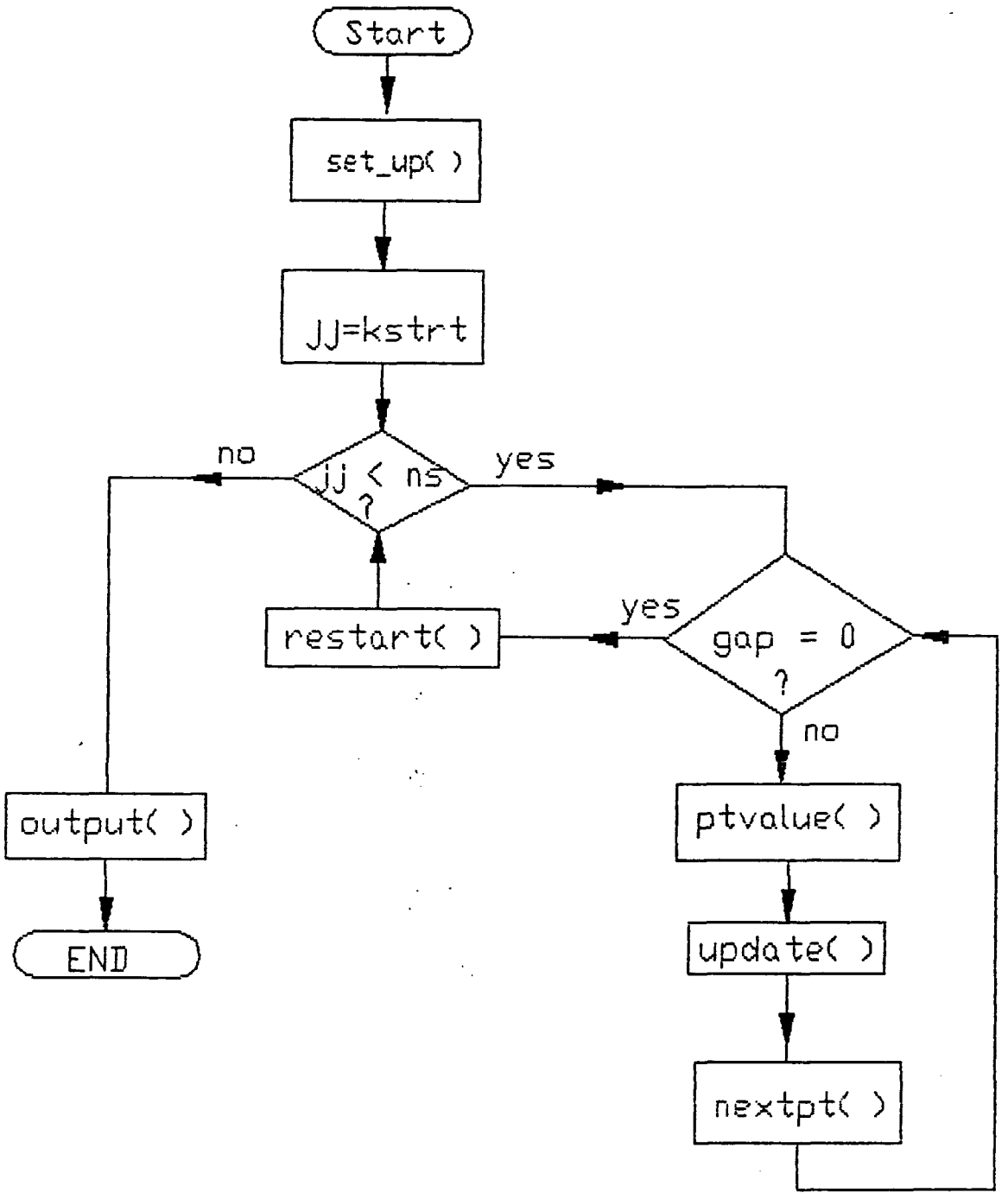


Fig. 4.1 - Flow Chart for Main( )

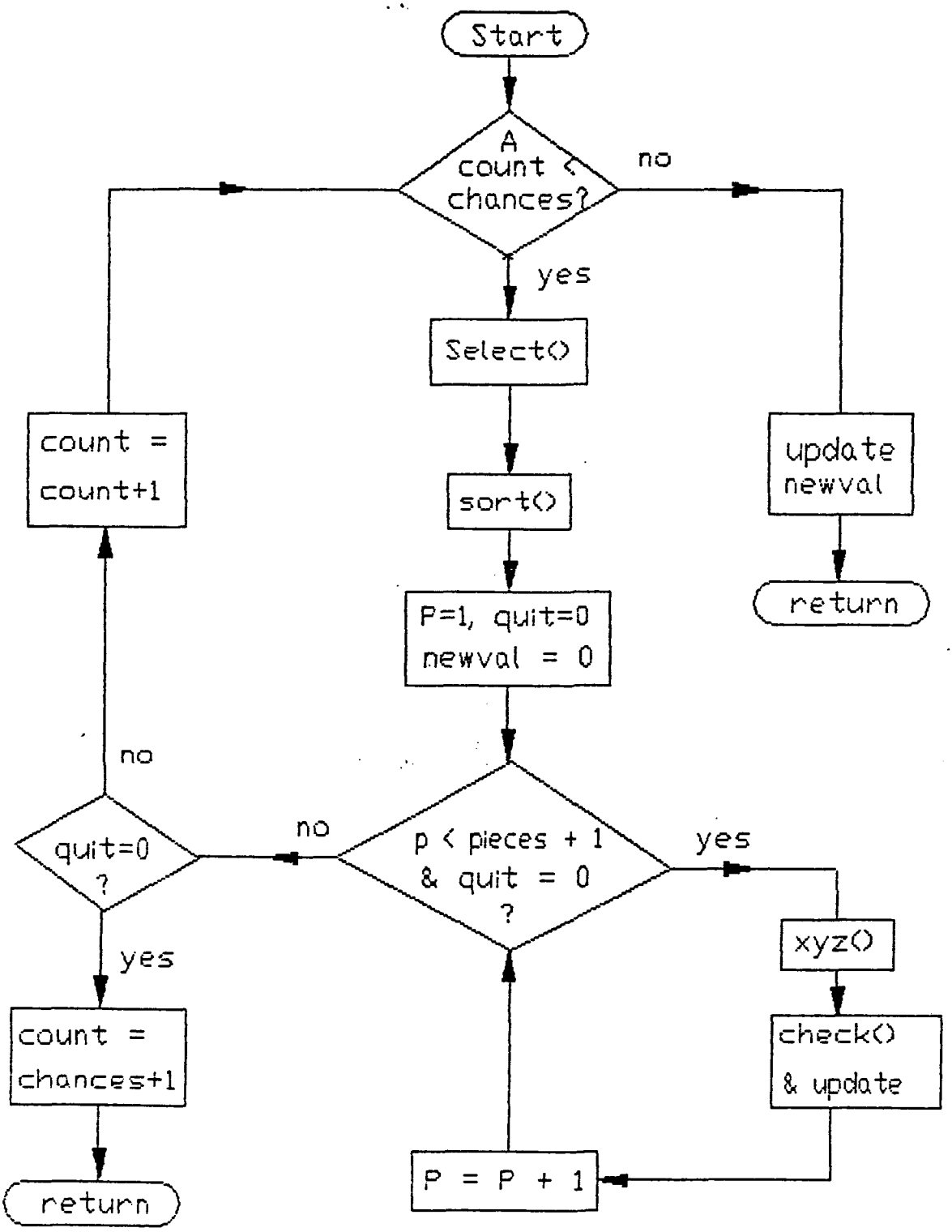


Fig 4.2 - Flow Chart for ptvalue()

References

1. Balas, E. & A. Ho, "Set Covering Algorithms Using Cutting Planes, Heuristics and Subgradient Optimization: A Computational Study", Mathematical Programming Study 12, (1980) pp. 37-60

2. Balas, E. & S.M.Ng, "On the Set Covering Polytope: II, Lifting The Facets With Coefficients in {0,1,2}" Math Programming, Vol. 45, No. 1 (1989) pp. 1-20

3. French, S., Sequencing and Scheduling, Ellis Horwood (1982)

4. Garfinkel, R.S. & G.L. Nemhauser, Integer Programming, John Wiley & Sons (1972)

5. Murty, K., Linear and Combinatorial Programming, John Wiley & Sons (1976)

6. Nobili, P. & A. Sassano, Facets and Lifting Procedures For The Set Covering Polytope" Math Programming, Vol. 45, No. 2 (1989) pp. 111-138

7. O'hEigeartaigh, M, J.K. Lenstra & A.H.G. Rinnooy Kan (Editors), Combinatorial Optimization: Annotated Bibliographies, John Wiley & Sons (1985)

8. Parker, R.G. & R.L. Rardin, Discrete Optimization Academic Press (1988)

9. Sassano, A., "On The Facial Structure of The Set Covering Polytope", Math Programming, Vol. 44, No. 2 (1989) pp. 181-202

```
 /* discrete line search heuristic main driver*/
#define CHANCES 1
#define BIG 5000
#define LIM 20
#define RIM 80
#include "malloc.h"
#include "stdlib.h"
#include "stdio.h"
void setup(void);
void nextpt(void);
void ptvalue(void);
void update(void);
void restart(void);
void output(void);
void binen(int);
int probe(int, int, int, int);
int space[5];
float sensor[2][LIM], deploy[5][RIM];
int subset[LIM],  index[LIM];
int ns, kstrt, jj, objflag;
int pnt[3], newpt, gap;
float value[3], newval, tempv;
float result1[5][LIM], aresult[5][LIM], cresult[5][LIM];
int l, w, h, pieces, seed;
char fname[6],  fname1[7], fname2[7];
main()
 {
 FILE *fptr, *fptr1, *fptr2;
 int j;
 printf("___________________________________________________\n\n");
 printf("The set of data for this procedure  exists as a DATA\n");
 printf("FILE. You are required to give the NAME of this file.\n\n");
 printf("You may not use more than 6 characters for this file\n");
 printf("name.\n");
 printf("___________________________________________________\n\n\n");
 printf("Please give the FILE NAME - no more than 6 characters --> ");
 fflush(stdin); scanf("%s", fname);  printf("\n\n");
 fname1[0]='s';  fname2[0]='d';
 for (j=0; j<6; j++)
   {
   fname1[j+1]=fname[j]; fname2[j+1]=fname[j];
   }
 fptr=fopen(fname, "rb");
 fread(space, sizeof(space), 1, fptr); fclose(fptr);
 fptr1=fopen(fname1, "rb");
 fread(sensor, sizeof(sensor), 1, fptr1); fclose(fptr1);
 fptr2=fopen(fname2, "rb");
 fread(deploy, sizeof(deploy), 1, fptr2); fclose(fptr2);
 setup();
 for ( jj = kstrt; jj < ns; jj++ )
   { nextpt();
     while ( gap != 0 )
      { ptvalue(); update(); nextpt(); }
     restart();
   }
 output();
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* initiates the line search iterations */
```

```
void setup(void)
{
int i, j, k, temp, temp1, sum;
l=space[0]/space[3] + 1;
w=space[1]/space[3] + 1;
h=space[2]/space[3] + 1; pieces=l*w*h;  seed=1;
ns=space[4];  sum=0;
for (j=0; j<ns; j++)
  { index[j]=sum; temp=sensor[1][j]; sum+=temp; }
printf("Please SELECT an objective for this surveillance problem\n\n");
printf("            1 -- To Minimise Maintenance Costs\n\n");
printf("            2 -- To Minimise Maintenance Task Time\n\n");
printf("         YOUR SELECTION --> ");
scanf("%d", &objflag); printf("\n\n");
printf("Would you guess the number of surveillance sensor mechanisms\n");
printf("that may suffice in covering the given space? If so, ENTER\n");
printf("the number here, OTHERWISE ENTER the number zero (0) -->  ");
scanf("%d", &k); printf("\n\n");
if (k==0 || k>ns-2) kstrt=ns/2; else kstrt=k;
jj=kstrt; temp=0;
for ( j=0; j<kstrt; j++) /*first left-point and corresponding subset*/
     { subset[j]=1; temp=2*temp+1; }
for ( j=kstrt; j<ns; j++) subset[j]=0;
ptvalue(); value[0]=newval; pnt[0]=temp; tempv=newval;
for (i=0; i<5; i++)
  for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
for ( j=0; j<(ns-kstrt); j++) subset[j]=0; /* first right-point and*/
for ( j=(ns-kstrt); j<ns; j++) subset[j]=1; /*corresponding subset*/
for ( j=kstrt; j<ns; j++) temp *= 2;
ptvalue(); value[2]=newval; pnt[2]=temp;
if (newval<tempv)
 { tempv=newval;
   for (i=0; i<5; i++)
     for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
 }
pnt[1]=(pnt[0]+pnt[2]); binen(pnt[1]);
ptvalue(); value[1]=newval;
for (i=0; i<5; i++)
  for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];
}

 /* ------------------------------------------------ */
 /* computes the next point and corresponding subset */
void nextpt(void)
 { int p0, p1, p2;
  p0=pnt[0]; p1=pnt[1]; p2=pnt[2];
  if (p1-p0 >= p2-p1)
   { if (p1 > p0 + 1) { newpt = (p1+p0)/2; gap=1; }
     else gap = 0;
   }
  if (p1-p0 < p2-p1)
   { if (p2 > p1 + 1) { newpt = (p1+p2)/2; gap=2; }
     else gap=0;
   }
  binen(newpt);
 }


/* ------------------------------ */
/* updates points and their values */
```

```
void update(void)
 { int i, j;
  if (newval < value[1] )
   { for (i=0; i<5; i++)
      for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];
     if (gap==1)
       { value[2]=value[1]; value[1]=newval;
         pnt[2]=pnt[1]; pnt[1]=newpt;
       }
     if (gap==2)
       { value[0]=value[1]; value[1]=newval;
         pnt[0]=pnt[1]; pnt[1]=newpt;
       }
   }
  else
   { if (gap==1)
      { value[0]=newval; pnt[0]=newpt; }
     if (gap==2)
      { value[2]=newval; pnt[2]=newpt; }
   }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* does a binary enumeration */
void binen(int p)
 { int xx, j;
 xx=p;
 for (j=0;  j<ns; j++) subset[j]=0;
 j=0;
 while ( xx > 1 && j < ns )
   { subset[j] = xx%2;
     xx = (xx-(xx%2))/2; j+=1;
   }
 subset[j]=1;
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* restarts the line search iterations */
void restart(void)
 {
  int i, j, k, p, p1, temp, flag;
  if (value[1]<tempv)
     { tempv=value[1];
       for (i=0; i<5; i++)
         for (j=0; j<ns; j++) aresult[i][j]=result1[i][j];
     }
  p=0;
  if (jj==ns-1)
    { for (j=0; j<ns; j++)
        { subset[j]=1; p=(2*p) + 1 ; }
      ptvalue();
      if (newval < tempv)
        { tempv = newval;
          for (i=0; i<5; i++)
            for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
        }
    }
  if (jj<ns-1)
    { p=0;
      for ( j=0; j<jj+1; j++) /* left-point and corresponding subset*/
```

```
        { subset[j]=1; p=2*p+1; }
      for ( j=jj+1; j<ns; j++) subset[j]=0;
     ptvalue(); value[0]=newval; pnt[0]=p; tempv=newval;
     for (i=0; i<5; i++)
       for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
     for ( j=0; j<(ns-(jj+1)); j++) subset[j]=0; /* right-point and    */
     for ( j=(ns-(jj+1)); j<ns; j++) subset[j]=1; /*corresponding subset*/
     for ( j=jj+1; j<ns; j++) p *= 2;
     ptvalue(); value[2]=newval; pnt[2]=p;
     if (newval<tempv)
      { tempv=newval;
        for (i=0; i<5; i++)
          for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
      }
     pnt[1]=(pnt[0]+pnt[2])/2;  binen(pnt[1]);
     ptvalue(); value[1]=newval;
     for (i=0; i<5; i++)
       for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];

    }
  }

/* ------------------------------- */
/* prints the output on the screen */
void output(void)
 { int i, j;
  char fn[4];
  if (objflag==1)
    { fn[0]='C'; fn[1]='O'; fn[2]='S'; fn[3]='T'; }
  if (objflag==2)
    { fn[0]='T'; fn[1]='I'; fn[2]='M'; fn[3]='E'; }
  if ( tempv < value[1] )
    { value[1]=tempv;
      for (i=0; i<5; i++)
       for (j=0; j<ns; j++) result1[i][j]=aresult[i][j];
    }
  printf("              OUTPUT  STATEMENT\n");
  printf("              ________________\n\n\n");
  printf(" Input FILE's name is '%s' \n\n\n", fname);
  printf(" Corresponding to %c%c%c%c value of %5.2f, the following\n",
           fn[0],fn[1], fn[2], fn[3], value[1]);
  printf(" table gives details of recommended surveillance sensor\n");
  printf(" mechanism deployments\n\n");
  printf("          ____________________________________________________ \n\n");
  printf("Sensor\tRadius\t\tLocation\t\tCost\t\tTime\n\n");
  for (j=0; j<ns; j++)
   { if ( result1[4][j] > -1 )
       printf("  %d\t%5.2f\t   (%5.2f, %5.2f, %5.2f)\t %4.2f\t      %5.2f\n",
      j+1, sensor[0][j],  result1[0][j], result1[1][j], result1[2][j],
      result1[3][j], result1[4][j]);
   }
 }
/* ------------------------------------------------------------   */
/* checks whether the given space may be covered by a given   */
/* set of sensor mechanisms, and, if so, computes the 'value' */
/* of the cover relative to given objective function          */
void select(void);
void sort(void);
void xyz(void);
void check(void);
```

```
int p, quit;
int pointer[LIM], slct[LIM], part[LIM];
float x, y, z, costime[LIM];
void ptvalue(void)
 { int j, count;                    /* selects deployment */
  count=0;
  while (count < CHANCES)
    { for (j=0; j<ns; j++)
        { cresult[0][j]=0; cresult[1][j]=0; cresult[2][j]=0;
          cresult[3][j]=0; cresult[4][j]=-2;  part[j]=0;
        }
      select(); sort(); p=1; quit=0; newval=0;
      while ( p < pieces+1 && quit == 0 )
        { xyz(); check();
          if (quit == 0) p+=1;
        }
      if (quit == 0) count=CHANCES+1;   /* a cover found */
      else count+=1;
    }
  if (quit==1)       /* not a cover */
    { newval=BIG;
      for (j=0; j<ns; j++)
        { cresult[0][j]=0; cresult[1][j]=0; cresult[2][j]=0;
          cresult[3][j]=0; cresult[4][j]=-2;
        }
    }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* selects a deployment of a given set of sensor mechanisms */
void select(void)
 { int j, r, num;
   for (j=0; j<ns; j++)
    { num=seed*7;  r=sensor[1][j];
      slct[j]=(subset[j])*(num%r); seed+=1;
    }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* sorts deployed sensor mechanisms by costs or by time */
void sort(void)
 { int i, j, k, col, temp1, count;
   float temp;
   if (objflag==1) k=3; else k=4;
   /* copying into costime */ i=0;
   for (j=0; j<ns; j++)
     { if (subset[j]==1)
        { col=index[j]+slct[j];
          costime[i]=deploy[k][col];
          pointer[i]=j; i+=1;
        }
     }
   /* sorting */ count=jj;
   while (count>0)
    { for (i=0; i<count; i++)
       { if (costime[i]>costime[i+1])
          { temp=costime[i]; costime[i]=costime[i+1];
            costime[i+1]=temp; temp1=pointer[i];
            pointer[i]=pointer[i+1]; pointer[i+1]= temp1;
          }
```

```
      }
    count-=1;
   }
 }

/* ---------------------------------------------------- */
/* computes the origin of the piece corresponding to p */
   void xyz(void)
   {
     int num, a, b, c, temp;
     num=p-1;   temp = l*w;
     c = num/temp;  num %= temp;
     b = num/l;     a = num%l;
     temp=space[3]; x=a*temp; y=b*temp; z=c*temp;
   }


/* ------------------------------------------ */
/* verifies that the point (x,y,z) is covered */
void check(void)
  { float a, b, c, r, sa, sb, sc, sr;
    int i, j, col, flag;
    flag=0; i=0;
    while (flag==0 && i<jj)
     { j=pointer[i]; col=index[j] + slct[j];
       r=sensor[0][j]; a=deploy[0][col];
       b=deploy[1][col]; c=deploy[2][col];
       sr=r*r; sa=(a-x)*(a-x); sb=(b-y)*(b-y); sc=(c-z)*(c-z);
       if ((sa+sb+sc) <= sr)
         { flag=1; cresult[0][j]=a; cresult[1][j]=b;
           cresult[2][j]=c; cresult[3][j]=deploy[3][col];
           cresult[4][j]=deploy[4][col];
           if (part[j]==0)
             { if (objflag==1) newval+=cresult[3][j];
               if (objflag==2 && newval<cresult[4][j])
                 newval=cresult[4][j];
               part[j]=1;
             }
         }
       i+=1;
    }
    if (flag==0) quit=1;
  }
```

APPENDIX III

SOURCE CODE FOR COMPUTER PROGRAM

```
/* A GENERALISED SET-COVERING PROCEDURE FOR SURVEILLANCE MAINTENANCE */
void datafls(void);
void search1(void);
void main(void)
 {
   char choice;
   menu:
   printf("\n\n                    MAIN  MENU \n\n");
   printf("     ____________________________________________ \n\n\n\n");
   printf("    SELECT \n\n\n");
   printf("      A -- To Create or View a Data File\n\n");
   printf("      B -- To Run this Space-covering Procedure\n");
   printf("           on an Existing Data File\n\n");
   printf("      C -- To Quit and Return to System\n\n\n");
   printf("    YOUR SELECTION --> ");  choice=getche();
   printf("\n\n");
   switch( choice )
     {
     case 'a':
     case 'A':
       datafls(); printf("\n\nPress ENTER to continue\n\n\n");
       getch(); goto menu;
     case 'b':
     case 'B':
       search1(); printf("\n\nPress ENTER to continue\n\n\n");
       getch(); goto menu;
     case 'c':
     case 'C':
       exit(0);
     default:
       goto menu;
     }
 }
/* @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
void w_data(void);
void v_data(void);
void datafls(void)
 {
   char choice;
   menu1:
   printf("\n\n           MENU  FOR  DATA  FILES\n");
   printf("_____________________________________________\n\n");
   printf("     SELECT\n\n");
   printf("        1 -- To Create a New File\n\n");
   printf("        2 -- To View an Existing File\n\n");
   printf("        3 -- To Return to MAIN Menu\n\n");
   printf("     YOUR SELECTION --> ");  choice=getche();
   printf("\n\n");
   switch( choice )
     {
     case '1':
       w_data(); printf("\n\nPress ENTER to continue\n\n\n");
       getch(); goto menu1;
     case '2':
       v_data(); printf("\n\nPress ENTER to continue\n\n\n");
       getch(); goto menu1;
     case '3':
       return;
     default:
```

```
        goto menu1;
      }
  }
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
#define LIM 20
#define RIM 80
#include "malloc.h"
#include "stdlib.h"
#include "stdio.h"
int space[5];
float sensor[2][LIM];
float deploy[5][RIM];
void spc(void);
void snsr(void);
void dply(void);
FILE *fptr, *fptr1, *fptr2;
char fname[6],  fname1[7], fname2[7];
void w_data(void)
   {
   int j;
   printf("___________________________________________________\n\n");
   printf("The set of data that you are about to enter will form\n");
   printf("a FILE. You are required to give a NAME to this file.\n\n");
   printf("You may not use more than 6 characters for the file\n");
   printf("name.\n");
   printf("___________________________________________________\n\n\n");
   printf("Please give the FILE NAME - no more than 6 characters --> ");
   fflush(stdin); scanf("%s", fname);   printf("\n\n");
   fname1[0]='s';  fname2[0]='d';
   for (j=0; j<6; j++)
     {
     fname1[j+1]=fname[j]; fname2[j+1]=fname[j];
     }

   spc(); snsr();  dply();
   }
/* information on space under surveillance*/
void spc(void)
   {
   int j, k;
   printf("____________________________________________________\n\n");
   printf(" You will be asked to provide information about the\n");
   printf(" space, S, under surveillance, and the potential intruder's\n");
   printf(" size. We shall assume that S is a rectangular volume \n");
   printf(" located at the origin,(0,0,0), of the 3-dimensional space.\n");
   printf("____________________________________________________\n\n");
   printf("Please give LENGTH, WIDTH, HEIGHT of S, l:w:h --> ");
   fflush(stdin);
   scanf("%d:%d:%d", &space[0], &space[1], &space[2]);
   printf("\n\n");
   printf("Please give SIZE OF POTENTIAL INTRUDER --> ");
   fflush(stdin);
   scanf("%d", &space[3]); printf("\n\n");
   printf("Please give TOTAL NUMBER OF AVAILABLE \n");
   printf("SURVEILLANCE SENSOR MECHANISMS --> "); fflush(stdin);
   scanf("%d", &space[4]); printf("\n\n\n\n");
   if ( (fptr = fopen(fname, "wb"))==NULL)
      { printf("Can't open %s n", fname); exit(); }
   fwrite(space, sizeof(space , 1, fptr);
   fclose(fptr);
```

```
 }
/* DATA ON SENSORS: SPAN RADIUS, AND DEPLOYMENT OPTIONS*/
void snsr(void)
 {
 int j;
  printf("__________________________________________________\n\n");
  printf("You will next supply  information about the coverage \n");
  printf("of each sensor mechanism -- the span, and the number of\n");
  printf("possible positioning or deployment of each sensor  \n");
  printf("mechanism.  Each sensor mechanism 'covers' a sphere\n");
  printf("whose centroid will play the role of deployment location.\n");
  printf("__________________________________________________\n\n\n");
  for (j=0; j<space[4]; j++)
    {
    printf("Please give the COVERAGE RADIUS of sensor mechanism\n");
    printf("number %d --> ", j+1); fflush(stdin);
    scanf("%f", &sensor[0][j]); printf("\n\n");
    printf("Please give the NUMBER OF POSSIBLE DEPLOYMENTS \n");
    printf("of sensor mechanism number %d --> ", j+1); fflush(stdin);
    scanf("%f", &sensor[1][j]); printf("\n\n");
    }
  if ( (fptr1 = fopen(fname1, "wb"))==NULL)
     { printf("Can't open %s\n", fname1); exit(); }
  fwrite(sensor, sizeof(sensor), 1, fptr1);
  fclose(fptr1);
 }
/*DATA ABOUT DEPLOYMENT OPTIONS*/
void  dply(void)
 {
  int j, k, m;
  printf("__________________________________________________\n\n");
  printf("You will next supply information concerning deployment\n");
  printf("locations, deployment costs, and the amount of time \n");
  printf("needed to complete each deployment as a task, that\n");
  printf("is, the amount of time needed to move each sensor\n");
  printf("mechanism from current location to deployment location.\n");
  printf("__________________________________________________\n\n\n");
  m=0;
  for (j=0; j<space[4]; j++)
    {
    for (k=0; k<sensor[1][j]; k++)
      {
      printf("Please give the  DEPLOYMENT LOCATION,  x,y,z  of the \n");
      printf("%d th deployment of sensor mechanism %d --> ", k+1, j+1);
      fflush(stdin);
      scanf("%f,%f,%f", &deploy[0][m], &deploy[1][m], &deploy[2][m]);
      printf("\n\n");
      printf("Please give the COST and DEPLOYMENT TASK TIME, c:t, \n");
      printf("of the %d th deployment of sensor mechanism %d --> ",
        k+1, j+1);  fflush(stdin);
      scanf("%f:%f", &deplc/[3][m], &deploy[4][m]); printf("\n\n");
      m+=1;
      }
    }
  if ( (fptr2 = fopen(fname2,  wb"))==NULL)
     { printf("Can't open  ile  please"); exit(); }
  fwrite(deploy, sizeof(de loy , 1, fptr2);
  fclose(fptr2);
 }
/* @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ )@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
```

```
void v_data(void)
 {
  int j, k, m;
  printf("_____\n\n");
  printf("The set of data that you are about to view exists as\n");
  printf("a FILE. You are required to give the NAME of this file.\n\n");
  printf("You may not use more than 6 characters for this file\n");
  printf("name.\n");
  printf("_____\n\n\n");
  printf("Please give the FILE NAME - no more than 6 characters --> ");
  fflush(stdin); scanf("%s", fname);   printf("\n\n");
  fname1[0]='s';  fname2[0]='d';
  for (j=0; j<6; j++)
    {
    fname1[j+1]=fname[j]; fname2[j+1]=fname[j];
    }
 fptr=fopen(fname, "rb");
 fread(space, sizeof(space), 1, fptr);
 fclose(fptr);
 printf("%s DATA ABOUT SPACE, S, UNDER SURVEILLANCE\n", fname);
 printf("_____\n\n");
 printf("Dimensions: %d, %d, %d\n\n", space[0], space[1], space[2]);
 printf("Size of potential intruder: %d\n\n\n", space[3]);
 printf("Press any key to continue file viewing\n\n\n");
 getch();
 printf("%s DATA ABOUT AVAILABLE SENSOR MECHANISMS\n", fname);
 printf("_____\n\n");
 fptr1=fopen(fname1, "rb");
 fread(sensor, sizeof(sensor), 1, fptr1);
 fclose(fptr1);
 printf("Number of sensor mechanisms: %d\n\n", space[4]);
 printf("sensor mechanism    radius of span    deployment options\n");
 for (j=0; j<space[4]; j++)
   printf("\t%d\t\t%f\t\t%f\n", j+1, sensor[0][j], sensor[1][j]);
 printf("\n\n\n");
 printf("Press any key to continue file viewing\n\n\n"); getch();
 printf("%s DATA ABOUT DEPLOYMENT OPTIONS\n", fname);
 printf("_____\n\n");
 fptr2=fopen(fname2, "rb");
 fread(deploy, sizeof(deploy), 1, fptr2);
 fclose(fptr2);  m=0;
 for (j=0; j<space[4]; j++)
  for (k=0; k<sensor[1][j]; k++)
   {
   printf("<--DEPLOYMENT OPTION %d of sensor mechanism %d-->\n\n", k+1, j+1);
   printf("Location: ( %f, %f, %f )\n\n", deploy[0][m], deploy[1][m],
              deploy[2][m]);
   printf("Cost: %f \t\t", deploy[3][m]);
   printf("Deployment task time: %f\n\n\n", deploy[4][m]);  m+=1;
   printf("Press any key to continue file viewing\n\n\n"); getch();
   }
}
/* @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
 /* discrete line search heuristic main driver*/
#define CHANCES 1
#define BIG 5000
#define LIM 20
#define RIM 80
#include "malloc.h"
#include "stdlib.h"
```

```
#include "stdio.h"
void setup(void);
void nextpt(void);
void ptvalue(void);
void update(void);                                  /* functions */
void restart(void);
void output(void);
void binen(int);
int probe(int, int, int, int);                     /* global variables */
int space[5];
float sensor[2][LIM], deploy[5][RIM];
int subset[LIM],  index[LIM];
int ns, kstrt, jj, objflag;
int pnt[3], newpt, gap;
float value[3], newval, tempv;
float result1[5][LIM], aresult[5][LIM], cresult[5][LIM];
int l, w, h, pieces, seed;
char fname[6],  fname1[7], fname2[7];
void search1(void)                                              /* beginning */
 {
 FILE *fptr, *fptr1, *fptr2;
 int j;
 printf("__________________________________________________\n\n");
 printf("The set of data for this procedure  exists as a DATA\n");
 printf("FILE. You are required to give the NAME of this file.\n\n");
 printf("You may not use more than 6 characters for this file\n");
 printf("name.\n");
 printf("__________________________________________________\n\n\n");
 printf("Please give the FILE NAME - no more than 6 characters --> ");
 fflush(stdin); scanf("%s", fname);  printf("\n\n");
 fname1[0]='s';  fname2[0]='d';
 for (j=0; j<6; j++)
   {
   fname1[j+1]=fname[j]; fname2[j+1]=fname[j];
   }
 fptr=fopen(fname, "rb");
 fread(space, sizeof(space), 1, fptr); fclose(fptr);
 fptr1=fopen(fname1, "rb");
 fread(sensor, sizeof(sensor), 1, fptr1); fclose(fptr1);
 fptr2=fopen(fname2, "rb");
 fread(deploy, sizeof(deploy), 1, fptr2); fclose(fptr2);
 setup();
 for ( jj = kstrt; jj < ns; jj++ )
   { nextpt();                                     /* see flow chart */
     while ( gap != 0 )
      { ptvalue(); update(); nextpt(); }
     restart();
   }
 output();
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* initiates the line search iterations */
 void setup(void)
 {
 int i, j, k, temp, temp1, sum;
 l=space[0]/space[3] + 1;
 w=space[1]/space[3] + 1;
 h=space[2]/space[3] + 1; pieces=l*w*h;  seed=3*kstrt+1; /* random seed */
 ns=space[4];  sum=0;
```

```
  for (j=0; j<ns; j++)
    { index[j]=sum; temp=sensor[1][j]; sum+=temp; }
  printf("Please SELECT an objective for this surveillance problem\n\n");
  printf("          1 -- To Minimise Maintenance Costs\n\n");
  printf("          2 -- To Minimise Maintenance Task Time\n\n");
  printf("       YOUR SELECTION --> ");
  scanf("%d", &objflag); printf("\n\n");
  printf("Would you guess the number of surveillance sensor mechanisms\n");
  printf("that may suffice in covering the given space? If so, ENTER\n");
  printf("the number here, OTHERWISE ENTER the number zero (0) -->  ");
  scanf("%d", &k); printf("\n\n");
  if (k==0 || k>ns-2) kstrt=ns/2; else kstrt=k;
  jj=kstrt; temp=0;
  for ( j=0; j<kstrt; j++) /*first left-point and corresponding subset*/
       { subset[j]=1; temp=2*temp+1; }
  for ( j=kstrt; j<ns; j++) subset[j]=0;
  ptvalue(); value[0]=newval; pnt[0]=temp; tempv=newval;
  for (i=0; i<5; i++)
    for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
  for ( j=0; j<(ns-kstrt); j++) subset[j]=0; /* first right-point and*/
  for ( j=(ns-kstrt); j<ns; j++) subset[j]=1; /*corresponding subset*/
  for ( j=kstrt; j<ns; j++) temp *= 2;
  ptvalue(); value[2]=newval; pnt[2]=temp;
  if (newval<tempv)
   { tempv=newval;
     for (i=0; i<5; i++)
       for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
   }
  pnt[1]=(pnt[0]+pnt[2])/2; binen(pnt[1]); sum=0;
  for (j=0; j<ns; j++) sum+=subset[j];   j=0;
  while(sum<jj)
    { sum+=1-subset[j]; subset[j]=1; j+=1; }
  while(sum>jj)
    { sum-=subset[j]; subset[j]=0; j+=1; }
  ptvalue(); value[1]=newval;
  for (i=0; i<5; i++)
    for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];
}

 /* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
 /* computes the next point and corresponding subset */
void nextpt(void)
 { int p0, p1, p2, j, sum;
   p0=pnt[0]; p1=pnt[1]; p2=pnt[2];
   if (p1-p0 >= p2-p1)
    { if (p1 > p0 + 1) { newpt = (p1+p0)/2; gap=1; }
      else gap = 0;
    }
   if (p1-p0 < p2-p1)
    { if (p2 > p1 + 1) { newpt = (p1+p2)/2; gap=2; }
      else gap=0;
    }
   binen(newpt);  sum=0;
   for (j=0; j<ns; j++) sum+=subset[j];   j=0;
   while(sum<jj)
     { sum+=1-subset[j]; subset[j]=1; j+=1; }
   while(sum>jj)
     { sum-=subset[j]; subset[j]=0; j+=1; }
   }
```

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* updates points and their values */
void update(void)
 { int i, j;
  if (newval < value[1] )
   { for (i=0; i<5; i++)
      for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];
     if (gap==1)
       { value[2]=value[1]; value[1]=newval;
         pnt[2]=pnt[1]; pnt[1]=newpt;
       }
     if (gap==2)
       { value[0]=value[1]; value[1]=newval;
         pnt[0]=pnt[1]; pnt[1]=newpt;
       }
   }
  else
   { if (gap==1)
      { value[0]=newval; pnt[0]=newpt; }
      if (gap==2)
      { value[2]=newval; pnt[2]=newpt; }
   }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* does a binary enumeration */
void binen(int p)
 { int xx, j;
 xx=p;
 for (j=0;  j<ns; j++) subset[j]=0;
 j=0;
 while ( xx > 1 && j < ns )
   { subset[j] = xx%2;
     xx = (xx-(xx%2))/2; j+=1;
   }
 subset[j]=1;
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* restarts the line search iterations */
void restart(void)
 {
  int i, j, k, p, p1, temp, flag, sum;
  if (value[1]<tempv)    /* save current best value */
     { tempv=value[1];
       for (i=0; i<5; i++)
         for (j=0; j<ns; j++) aresult[i][j]=result1[i][j];
     }
  p=0;
  if (jj==ns-1)   /* if full set is to picked up next */
    { for (j=0; j<ns; j++)
        { subset[j]=1; p=(2*p) + 1 ; }
      ptvalue();
      if (newval < tempv)
        { tempv = newval;
          for (i=0; i<5; i++)
            for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
        }
    }
  if (jj<ns-1)
```

```
    { p=0;
      for ( j=0; j<jj+1; j++) /* left-point and corresponding subset*/
        { subset[j]=1; p=2*p+1; }
      for ( j=jj+1; j<ns; j++) subset[j]=0;
      ptvalue(); value[0]=newval; pnt[0]=p; tempv=newval;
      for (i=0; i<5; i++)
        for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
      for ( j=0; j<(ns-(jj+1)); j++) subset[j]=0; /* right-point and    */
      for ( j=(ns-(jj+1)); j<ns; j++) subset[j]=1; /*corresponding subset*/
      for ( j=jj+1; j<ns; j++) p *= 2;
      pnt[2]=p; ptvalue(); value[2]=newval;
      if (newval<tempv)
       { tempv=newval;
         for (i=0; i<5; i++)
           for (j=0; j<ns; j++) aresult[i][j]=cresult[i][j];
       }
      pnt[1]=(pnt[0]+pnt[2])/2;  binen(pnt[1]); sum=0;
      for (j=0; j<ns; j++) sum+=subset[j];   j=0;
      while(sum<jj+1)
        { sum+=1-subset[j]; subset[j]=1; j+=1; }
      while(sum>jj+1)
        { sum-=subset[j]; subset[j]=0; j+=1; }
      ptvalue(); value[1]=newval;
      for (i=0; i<5; i++)
        for (j=0; j<ns; j++) result1[i][j]=cresult[i][j];
    }
  }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* prints the output on the screen */
void output(void)
 { int i, j;
  char fn[4];
  if (objflag==1)
    { fn[0]='C'; fn[1]='O'; fn[2]='S'; fn[3]='T'; }
  if (objflag==2)
    { fn[0]='T'; fn[1]='I'; fn[2]='M'; fn[3]='E'; }
  if ( tempv < value[1] )
    { value[1]=tempv;
      for (i=0; i<5; i++)
       for (j=0; j<ns; j++) result1[i][j]=aresult[i][j];
    }
  printf("              OUTPUT  STATEMENT\n");
  printf("              ________________\n\n\n");
  printf(" Input FILE's name is '%s' \n\n\n", fname);
  printf(" Corresponding to %c%c%c%c value of %5.2f, the following\n",
           fn[0],fn[1], fn[2], fn[3], value[1]);
  printf(" table gives details of recommended surveillance sensor\n");
  printf(" mechanism deployments\n\n");
  printf("      ____________________________________________________ \n\n");
  printf("Sensor\tRadius\t\tLocation\t\tCost\t\tTime\n\n");
  for (j=0; j<ns; j++)
   { if ( result1[4][j] > -1 )
      printf("  %d\t%5.2f\t   (%5.2f, %5.2f, %5.2f)\t %4.2f\t      %5.2f\n",
     j+1, sensor[0][j],  result1[0][j], result1[1][j], result1[2][j],
     result1[3][j], result1[4][j]);
   }
 }
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* checks whether the give  space may be covered by a given   */
```

```
/* set of sensor mechanisms, and, if so, computes the 'value' */
/* of the cover relative to given objective function          */
void select(void);
void sort(void);
void xyz(void);
void check(void);
int p, quit;
int pointer[LIM], slct[LIM], part[LIM];
float x, y, z, costime[LIM];
void ptvalue(void)
 { int j, count;                    /* selects deployment */
  count=0;
  while (count < CHANCES)
    { for (j=0; j<ns; j++)
        { cresult[0][j]=0; cresult[1][j]=0; cresult[2][j]=0;
          cresult[3][j]=0; cresult[4][j]=-2;  part[j]=0;
        }
      select(); sort(); p=1; quit=0; newval=0;
      while ( p < pieces+1 && quit == 0 )
        { xyz(); check();
          if (quit == 0) p+=1;
        }
      if (quit == 0) count=CHANCES+1;    /* a cover found */
      else count+=1;
    }
  if (quit==1)       /* not a cover */
    { newval=BIG;
      for (j=0; j<ns; j++)
        { cresult[0][j]=0; cresult[1][j]=0; cresult[2][j]=0;
          cresult[3][j]=0; cresult[4][j]=-2;
        }
    }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* selects a deployment of a given set of sensor mechanisms */
void select(void)
 { int j, r, num;
   for (j=0; j<ns; j++)
    { num=seed*7;  r=sensor[1][j];
      slct[j]=(subset[j])*(num%r); seed+=1;
    }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* sorts deployed sensor mechanisms by costs or by time */
void sort(void)
 { int i, j, k, col, temp1, count;
   float temp;
   if (objflag==1) k=3; else k=4;
   /* copying into costime */ i=0;
   for (j=0; j<ns; j++)
     { if (subset[j]==1)
         { col=index[j]+slct[j];
           costime[i]=deploy[k][col];
           pointer[i]=j; i+=1;
         }
     }
   /* sorting */ count=jj;
   while (count>0)
```

```
    { for (i=0; i<count; i++)
       { if (costime[i]>costime[i+1])
          { temp=costime[i]; costime[i]=costime[i+1];
            costime[i+1]=temp; temp1=pointer[i];
            pointer[i]=pointer[i+1]; pointer[i+1]= temp1;
          }
       }
      count-=1;
     }
 }

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* computes the origin of the piece corresponding to p */
   void xyz(void)
   {
     int num, a, b, c, temp;
     num=p-1;    temp = l*w;
     c = num/temp;  num %= temp;
     b = num/l;      a = num%l;
     temp=space[3]; x=a*temp; y=b*temp; z=c*temp;
   }


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* verifies that the point (x,y,z) is covered */
void check(void)
  { float a, b, c, r, sa, sb, sc, sr;
    int i, j, col, flag;
    flag=0; i=0;
    while (flag==0 && i<jj)
     { j=pointer[i]; col=index[j] + slct[j];
       r=sensor[0][j]; a=deploy[0][col];
       b=deploy[1][col]; c=deploy[2][col];
       sr=r*r; sa=(a-x)*(a-x); sb=(b-y)*(b-y); sc=(c-z)*(c-z);
       if ((sa+sb+sc) <= sr)
         { flag=1; cresult[0][j]=a; cresult[1][j]=b;
           cresult[2][j]=c; cresult[3][j]=deploy[3][col];
           cresult[4][j]=deploy[4][col];
           if (part[j]==0)
             { if (objflag==1) newval+=cresult[3][j];
               if (objflag==2 && newval<cresult[4][j])
                 newval=cresult[4][j];
               part[j]=1;
             }
         }
       i+=1;
    }
    if (flag==0) quit=1;
  }
```

APPENDIX IV

TEST PROBLEMS

FILE 1

		1	2	3	4	5	6
Radius of Sensor Span		5	5	5	5	5	5
No. of Optional Deployments		1	2	1	1	2	1
Deployment Locations	x	0	2	6	6	4	4
	y	0	0	2	0	4	4
	z	0	0	0	2	0	4
	x		4			2	
	y		0			0	
	z		4			0	
Cost		4	5.5	4	4	4.5	5
Time		3	4,5	5	6	5,4	5

Computed Solution:

COST: 18
1; (2,2); 3; 6

TIME: 5
1; (2,2); 3; 6

FILE 2

	1	2	3	4	5	6
Radius of Sensor Span	6	4	6	4	6	4
No. of Optional Deployments	1	2	1	1	2	1
Deployment x	0	2	6	6	4	4
Deployment y	0	0	2	0	4	4
Deployment z	0	0	0	2	0	4
Locations x		4			2	
Locations y		0			0	
Locations z		4			0	
Cost	4	5,5	4	4	4,5	5
Time	3	4,5	5	6	5.4	5

Computed Solution:

COST: 8
1; 3

TIME: 4
1; (5,2)

FILE 3

		1	2	3	4	5	6
Radius of Sensor Span		5	4	5	4	5	4
No. of Optional Deployments		1	2	1	1	2	1
Deployment Locations	x	0	2	6	6	4	4
	y	0	0	2	0	4	4
	z	0	0	0	2	0	4
	x		4			2	
	y		0			0	
	z		4			0	
Cost		4	5,5	4	4	4,5	5
Time		3	4,5	5	6	5,4	5

Computed Solution:

COST: 17
1; 4; (5,1); 6

TIME: 5
1; 2; 3; 6

FILE 4

		1	2	3	4	5	6
Radius of Sensor Span		5	5	4	5	4	4
No. of Optional Deployments		2	1	1	2	1	1
Deployment Locations	x	0	2	4	4	6	6
	y	0	0	4	4	2	0
	z	0	0	4	0	0	2
	x	2			4		
	y	0			0		
	z	0			4		
Cost		4,5	5	5	4,5	4	4
Time		3,4	4	5	5,5	5	6

Computed Solution:

COST: 18
(1,1); 2; 3; 6

TIME: 5
(1,1); 2; 3

FILE 5

		1	2	3	4	5	6
Radius of Sensor Span		4	4	6	4	6	6
No. of Optional Deployments		2	1	1	2	1	1
Deployment Locations	x	0	2	4	4	6	6
	y	0	0	4	4	2	0
	z	0	0	4	0	0	2
	x	2			4		
	y	0			0		
	z	0			4		
Cost		4,5	5	5	4,5	4	4
Time		3,4	4	5	5,5	5	6

Computed Solution:

COST: 13
(1,1); 3; 6

TIME: 5
(1,1); 2; 3

FILE 6

		1	2	3	4	5	6
Radius of Sensor Span		4	6	4	6	4	6
No. of Optional Deployments		1	2	1	1	2	1
Deployment Locations	x	0	2	6	6	4	4
	y	0	0	2	0	4	4
	z	0	0	0	2	0	4
	x		4			2	
	y		0			0	
	z		4			0	
Cost		4	5,5	4	4	4.5	5
Time		3	4,5	5	6	5,4	5

Computed Solution:

COST: 13
1; (2,1); 3

TIME: 5
1; (2,2); 6

FILE _7_____

		1	2	3	4	5	6
Radius of Sensor Span		4	5	4	5	4	5
No. of Optional Deployments		1	2	1	1	2	1
Deployment Locations	x	0	2	6	6	4	4
	y	0	0	2	0	4	4
	z	0	0	0	2	0	4
	x		4			2	
	y		0			0	
	z		4			0	
Cost		4	5,5	4	4	4,5	5
Time		3	4,5	5	6	5,4	5

Computed Solution:

COST: 18
(1,1); 2; 3; 6

TIME: 5
(1,1), 2; 6

FILE 8

		1	2	3	4	5	6
Radius of Sensor Span		4	4	5	4	5	5
No. of Optional Deployments		2	1	1	2	1	1
Deployment Locations	x	0	2	4	4	6	6
	y	0	0	4	4	2	0
	z	0	0	4	0	0	2
	x	2			4		
	y	0			0		
	z	0			4		
Cost		4,5	5	5	4,5	4	4
Time		3,4	4	5	5,5	5	6

Computed Solution:

COST: 14
(1,1); 2; 3

TIME: 5

FILE 9

		1	2	3	4	5	6
Radius of Sensor Span		4	4	4	4	4	4
No. of Optional Deployments		2	1	1	2	1	1
Deployment Locations	x	0	2	4	4	6	6
	y	0	0	4	4	2	0
	z	0	0	4	0	0	2
	x	2			4		
	y	0			0		
	z	0			4		
Cost		4,5	5	5	4,5	4	4
Time		3,4	4	5	5,5	5	6

Computed Solution:

FILE 10

		1	2	3	4	5	6
Radius of Sensor Span		6	6	4	6	4	4
No. of Optional Deployments		2	1	1	2	1	1
Deployment Locations	x	0	2	4	4	6	6
	y	0	0	4	4	2	0
	z	0	0	4	0	0	2
	x	2			4		
	y	0			0		
	z	0			4		
Cost		4,5	5	5	4,5	4	4
Time		3,4	4	5	5,5	5	6

COST: 5
(1,2)

Computed Solution:

TIME: 4
(1,2)